An Integrated Software Environment to Design Polymorphic Fault Tolerant Processors on Radiation Hardened FPGAs

Award Number: NNG06GE54G

PI: Dr. Aravind Dasu, Utah State University
L304E, ECE department
USU, Logan-84322-4120
dasu@engineering.usu.edu

Year 2 Report

Year 2 Achievements in Summary

1. A comprehensive literature review has been conducted to understand the current status of on-board autonomous mission planning based iterative repair algorithms, application-specific processor techniques, and use of FPGAs as on-board computers in the space environment.
2. An application-specific hardware architecture (pipelined processor) has been designed and developed for accelerating Iterative Repair algorithms. Initial performance tests indicate a speed up of 800 times or more when the custom architecture is compared with a widely used embedded PowerPC processor as well as a commodity desktop microprocessor.
3. A custom hardware compiler, termed the SATH (Simulated Annealing To Hardware) tool, has been developed to translate Iterative Repair C code into a custom pipelined hardware processor. This tool consists of several stages:
    a. An intermediate representation generator, which translates GCC intermediate code into an optimized, custom intermediate representation
    b. Constant extractors, which parse the source code for important parameters
    c. Design Space Explorers, which derive efficient architectures of pipelined stages based upon source code complexity and FPGA resource constraints
    d. An architecture-to-VHDL translator, which reduces the final architecture specification to a set of synthesizable VHDL files.
4. Publications:
    a. "A Coarse-grain Pipelined Architecture for Accelerating Iterative Repair-Type Event Scheduling on SRAM-FPGAs", Jonathan Phillips, Aravind Dasu. Submitted to the IEEE Transactions on VLSI, July 2007. Recommended for revision and re-submission within 21 days. It has been revised and resubmitted on Oct 9th.
    b. "Deriving FPGA based custom soft-core microprocessors for Mission Planning Algorithms", A. Dasu, J.D. Phillips. Proceedings of the 21st annual AIAA Small Satellite conference. Aug. 2007.
    c. "An ASIP architecture framework to facilitate automated design space exploration and synthesis for Iterative Repair solvers", A. Dasu, J.D. Phillips. Proceedings of the NASA Science and Technology Conference (NSTC) 2007.
5. Student involvement: 3 graduate students (2 MS + 1 PhD). The PhD student will defend his dissertation in the Spring semester of 2008.
6. Special session at the ERSA conference (engineering of reconfigurable systems and algorithms): I have been invited by the chair of ERSA conference (Dr. Toomas Plaks) to host a special session. I am focusing on a theme related to FPGAs for space based applications, algorithms and tools. Through this process I am trying to actively engage NASA groups from JPL, LaRC etc. to showcase their work at this conference.
7. Planning for Year 3: We have started preliminary work on fault tolerance and mitigation circuit design techniques and evaluation against a commodity TMR tool from Xilinx.

Abstract

Autonomous dynamic event scheduling, using Iterative Repair techniques such as those employed by remote agent software, is clearly becoming an essential component of space missions, as it enables spacecraft to adaptively schedule tasks in a dynamic, real-time environment. Through several missions such as Deep Space 1, TechSat-21 and Autonomous Sciencecraft Experiment it has been shown that there is great potential for significantly increasing scientific discovery with the help of AI-based autonomous on-board software such as CASPER. It is expected that many future missions such as Distributed spacecraft, MISUS (multi-rover integrated science understanding system) for planetary exploration and integrated human-robotic explorations, etc. will use even more powerful and complex AI-based mission planning, scheduling and replanning software/algorithms. Event rescheduling is a compute-intensive process. Typical applications involve scheduling hundreds of events that share tens or hundreds of resources. Iterative Repair problems are generally solved using combinatorial search heuristic methods, such as simulated annealing, genetic algorithms, or stochastic beam search. These heuristics are computationally intensive since they operate by gradually improving an initial solution over hundreds or thousands of iterations. In the past, the constraints of traditional computing platforms for spacecraft have precluded the realization of fast and efficient hardware to accelerate such algorithms. The recently proven viability of using SRAM-based FPGAs in space, however, provides an opportunity to design very low power, efficient, and fault-mitigating circuits customized to accelerate this class of autonomous-scheduling algorithms by one or more orders of magnitude. However, the design of such on-chip architectures is quite complicated and presently requires expertise in VLSI design, which makes using FPGAs user-unfriendly and complicated for software oriented mission engineers and scientists. Therefore, through this project we have developed a custom compiler that compiles ANSI-C code of a wide class of Iterative Repair algorithms into highly optimized low level circuits that can be mapped onto commodity FPGAs. As part of this compiler tool development, we first designed and implemented a template VLSI architecture for a reasonably complex mission-scheduling problem. We then compared the performance of this circuit (on an FPGA) to that of a widely used embedded PowerPC microprocessor as well as a commodity desktop microprocessor. Results showed a speedup of two orders and one order of magnitude respectively. We then followed up with the design and development of a new silicon-compiler tool flow that uses the template architecture in conjunction with a set of new design space exploration algorithms to deliver fully synthesizable VHDL code that can be readily mapped onto commodity Xilinx FPGAs.

TABLE OF CONTENTS

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are becoming increasingly popular as a platform of choice for spacecraft computer systems. FPGA-based designs are much cheaper and have a shorter development cycle than traditional Application-Specific Integrated Circuits (ASICs), and provide more computing power and efficiency than standard microprocessors. A sub-set of current and planned NASA missions that utilize FPGA technology include MARTE (Mars Astrobiology Research and Technology Experiment) [1] and the Discovery and New Frontier programs [2].

Simulated annealing is a widely used heuristic algorithm to solve challenging optimization problems. While it has been used extensively for static time design optimization, there is an increasing need to deploy such solvers in real time embedded systems. For example, NASA uses the CASPER and ASPEN tools [3] to design code for Iterative Repair, a simulated annealing algorithm that derives complicated event schedules on-board spacecraft. The complexity of these algorithms can be daunting for space based computers, which are significantly slower than state of the art microprocessors.

Combining FPGAs with simulated annealing algorithms would greatly improve system performance. Unfortunately, hardware architecture design targeting FPGAs is much harder and more time consuming than software design and is daunting for software engineers without expertise in VLSI design.
To mitigate this design flow barrier, a methodology and tool flow for the automatic derivation, from source C code of simulated annealing scheduling algorithms, of FPGA-based application-specific processors is presented. This tool flow is termed SATH (Simulated Annealing to Hardware). The rest of this document is organized as follows: A history of related works is presented with critical analysis in chapter 2. Chapter 3 presents the architecture template. Chapter 4 presents two new algorithms/techniques for resource estimation and scheduling. Chapter 5 describes the architecture derivation tool through the use of a template. Finally in Chapter 6 we present some results obtained accompanied by an analysis.

2. BACKGROUND

2.1 Significance

This project proposes and develops a methodology for deriving application specific processors for event scheduling algorithms in the space environment from high-level source code. This project embraces concepts from the fields of compilers, custom processor design, and C-to-gates translators. Deriving hardware from software is an inherently difficult procedure. Several attempts with varying levels of success have been performed over the past 20 years. Most products result in supporting only a restricted subset of C with additional compiler directives to give the tool hints on how to derive hardware.
The aim of this research is to create an application-specific C-to-architecture tool flow that will enable NASA mission planning engineers to efficiently utilize FPGAs as processing platforms for on-board mission planning and scheduling operations.

2.2 Field Programmable Gate Arrays

An FPGA is a silicon device that is in some ways comparable to an Application-Specific Integrated Circuit (ASIC). The vast majority of large-scale integrated devices used today in industry are ASICs, meaning that they are permanently configured for a specific application. An FPGA, on the other hand, consists of several static random access memory (SRAM) based reprogrammable logic blocks along with reprogrammable inter-block connections, allowing a single FPGA to be reprogrammed and used in a variety of applications. While FPGAs have the benefit of being dynamically reconfigurable, they are also larger, slower, and more power-hungry than ASICs. As FPGA technology continues to improve, they will command an increasing share of the integrated circuit market. Much research is currently being done on FPGA methods and applications.

2.3 Previous Work

Deriving architectures from high-level source code is a task that must borrow concepts and methods from several distinct research areas. For example, several efforts have been made in the past 20 years, with varying levels of success, to create tools for the translation of C source code into hardware specifications, with the aim of improving performance over that obtained on general-purpose machines. Similar efforts have been made in the field of parallel compilers, in which sequential high-level code is partitioned across several processing units to improve program performance. Other related areas include the development of Application-Specific Instruction Processors (ASIPs) and designing radiation-hardened FPGAs for use in the space environment.

2.3.1 C to Hardware

Converting C code into a hardware specification has been a much-researched area over the past two decades. Several tools have been produced over this time period, each of which targets specific areas of the software to hardware conversion. In [4] a

comprehensive summary of the different projects is provided. A summary of each of these tools is provided here.

The first attempt at a C to Gates tool was Cones [5], which was developed in 1988 at AT&T Bell Laboratories. Cones works on a subset of C with the introduction of some additional directives to facilitate translation to hardware. Cones translates C functions into combinatorial blocks, based upon the premise that a function consists of several inputs that influence one output. Arithmetic and logic statements are reduced through the use of Karnaugh Maps and similar techniques. Cones also unrolls simple loops. Cones cannot handle pointers, nested loops, recursive function calls, or dynamic memory allocation. Additional syntax is introduced to specify input and output data for each C function.

HardwareC [6] was developed at Stanford University in 1990. It is based upon the same syntax as conventional C, but includes custom hardware semantics. HardwareC additions include both procedural and declarative semantics, which means a design can consist of either sequences of operations or as a structure of interconnecting components. HardwareC models hardware as concurrent processes with inter-process communication facilities. The level of parallelism can be specified by the programmer to be sequential, data-parallel, or parallel for a given design. Lastly, HardwareC supports constraint specifications, where time and resource constraints can be imposed.

Transmogrifier C [7] from the University of Toronto was released in 1995. It is another variation on a subset of the C programming language. Integer addition and subtraction are supported. Compiler directives are used to specify data lengths. If statements, while loops, and function calls are also supported. The compiler does not support multiplication, division, pointers, arrays, structures, recursion, or floating-point arithmetic.

In 2002 SystemC [8] was introduced. SystemC is actually a C++ library. Classes are defined for simple Verilog constructs such as combinatorial and sequential modules. Simple, user-specified concurrency is allowed through the use of threads. SystemC is primarily a simulation language, although a restricted subset can be synthesized. Similarly, Ocapi [9] and PDL++ [10] are also built on C++. Proprietary classes are provided for creating finite state machines and data paths. The C++ code is translated to VHDL or Verilog for synthesis and design implementation.

One of the most successful ventures at translating C code to hardware was C2Verilog [11], which was introduced in 1998. C2Verilog supports standard ANSI C code. The programmer is not required to provide any hints or directives on concurrency or partitioning. The entire C language is supported, including pointers, recursion, floating-point data types, arrays, structures, etc. C2Verilog works by performing global analysis on a program and creating always blocks and concurrent statements. Functions are represented using state machines. Pointers and dynamic memory allocation are managed by creating a dedicated RAM of sufficient size and sufficient IO ports to allow for efficient access.

Cyber [12], released by NEC in 1999, is yet another subset of the C programming language. The C subset is once again augmented with bit-length and input/output port declarations, as well as data-transfer types such as registers, terminals, latches, and tri-state buffers. The programmer must also specify code sections that are synchronous, asynchronous, or concurrent. Typically, difficult constructs such as recursion, dynamic memory allocation, and pointers are not supported. This augmented subset of C is termed BDL (Behavior Description Language). Cyber is actually a behavioral synthesis system that takes behavioral code in BDL or VHDL and produces synthesizable register transfer logic.

Handel-C [13], developed by Celoxica, is a widely-used C variant. Handel-C is built upon the CSP [14] algebra for modeling process concurrency. Data widths can be specified by the programmer. The programmer can designate code sections as being concurrent. Input and output ports are introduced to allow an interface with the outside world. Synchronous message massing between processes is available. A variation on the basic C switch statement generates hardware multiplexers. Every instruction in Handel-C takes a single clock cycle. Bach C [15] is almost identical to Handel-C, providing the same explicit concurrency and message-passing capabilities. Pointers are not handled in either Handel-C or Bach C.

Another C variant is SpecC [16]. SpecC provides a set of 33 key words as additions to the ANSI C language that specify how the compiler should create finite state machines, concurrency, pipelining, etc. Once again, complicated features such as recursion and pointers are not supported.

The Trident C compiler [17] is a tool targeted specifically at floating-point operations. Concurrency and throughput are maximized through the use of pipelined floating-point units on an FPGA. Yet again, features such as dynamic memory allocation and pointer manipulations are not supported.

Another work worthy of mention is SPARK [18]. SPARK translates a subset of ANSI C to register-transfer level VHDL. Much work is done on the analysis of loops and conditional branches. Speculative code execution is also employed, meaning that computations performed within a conditional block are commenced before it has been determined that the block will be entered. Compile-time analysis is performed to determine what the best instructions are for speculative execution. The SPARK compiler cannot handle pointers or dynamic memory allocations.

The final system of note is CASH [19]. CASH, or Compiler for Application-Specific Hardware, differs from all previously mentioned systems because it generates asynchronous hardware. Starting from a pure ANSI-C code, CASH identifies instruction level parallelism and generates asynchronous data-flow circuits that support these parallel constructs.

To summarize this discussion on work in the area of C-to-architecture translation, compiling pure C code into gates is a difficult task. Common areas of difficulty include recursive function calls, dynamic memory allocation, pointer manipulation, support for all C data types (including floating point representations), and detection of concurrency. In [20] a detailed discussion of the shortcomings of using C to derive hardware is presented. Specifically, the concepts of concurrency, timing, data types, and communication can not be specified using standard C. These issues are generally resolved by either reducing the input language to a restricted subset of C, or by introducing compiler directives or other keywords or labels to indicate how the compiler should proceed. The ideal situation is to start with fully compliant ANSI C source code, thus allowing for the translation of existing code directly into hardware without modification. Of all the projects discussed, only C2Verilog claims to support the entire ANSI C standard with no restrictions or additions. This means that the C2Verilog compiler is responsible for all extraction of parallelism. While the C2Verilog compiler can identify instruction and loop-level parallelism, it cannot extract process-level parallelism. The ability of the compiler to recognize parallelism is directly dependent upon the syntax of the source code – source codes written with concurrency in mind generally result in better hardware architectures than those that are written for conventional sequential machines. In addition, current C-to-hardware tools fail to take into account several important hardware design factors, including available FPGA area, power consumption, execution speed, and fault tolerance, all of which are critical to engineering in the space environment.

2.3.2 Parallel Compilers

Parallel compilers are used to partition high-level sequential code across multiple processors. The high-level programming paradigm that has been followed for decades is based upon sequential code executing on a sequential processor. Retargeting this sequential code for exploitation by parallel processors in an efficient manner has been a difficult task. Parallelism can be extracted at multiple levels. Instruction level parallelism, basic-block level parallelism, loop level parallelism, and function level parallelism are some examples. Different compilers take different approaches in the manner in which the parallelization occurs.

Several compilers have been developed which extract parallelism at either a fine-grained or a coarse-grained level. For example, ICU-PFC [21] is a FORTRAN compiler that targets loop-level parallelism. Control flow analysis is performed on the source code to derive loop structures. Data flow analysis is performed to determine the presence or absence of inter-loop dependencies. FORTRAN augmented with concurrent constructs is the result. As FORTRAN is generally used for scientific and other compute-intensive applications, performing loop-level parallelization analysis is a reasonable method. There is a potential for significant data dependencies to exist between loops, however, which would significantly restrict the level of improvement. Performing additional analysis at either instruction level, basic-block level, or function level may alleviate some of these restrictions.

Another compiler worthy of mention is OSCAR [22]. OSCAR is a multigrain FORTRAN compiler that attempts to embrace all levels of parallelism from instruction level to program level. Extensive control- and data-flow analysis is performed to determine which levels of parallelism exploitation will yield maximum improvement. Efficient FORTRAN code is produced that generally performs between two and four times better than code produced by less-rigorous compilers.

Maybe the most famous of parallel compilers is SUIF [23], developed at Stanford University. SUIF is designed to extract coarse-grained parallelism, specifically at the function, loop, and independent memory access levels. The goal of extracting coarse-level parallelism, as opposed to instruction-level parallelism, is to provide large pieces of code that can be executed on different processors with minimal inter-processor communication or synchronization. The main challenge faced by SUIF and similar compilers is in predicting memory access patterns, especially when arrays are accessed with run-time-generated indices. SUIF organizes memory storage to exploit areas of potential parallelization. Work continues on SUIF to further refine its parallelism extraction capabilities.

A slightly different approach to parallel compilation is taken by the RAW architecture [24]. RAW consists of a set of 16 tiled processors connected via a high-bandwidth network, all contained in a single chip. The RAW-specific compiler attempts to efficiently map applications to processing tiles, extracting parallelism at both the instruction and thread levels. In order to achieve maximum system throughput, however, it is conceded that hand optimization is needed.

These types of compilers are generally designed for specific target architectures. The properties of the architecture drive the optimizations performed by the compiler. Programs with minimal control flow work best for parallelization, as less synchronization is needed. Developing an efficient parallel compiler front end that can be mapped to several target processors is a difficult task.

2.3.3 Design Space Exploration

Design space exploration in the context of FPGA-based architectures is a powerful tool. Exploring a design space is, in essence, searching the combinatorial space of all possible hardware architectures that can support a given function. The goal is to identify the architecture that yields the best tradeoff between conflicting goals, such as FPGA area usage versus system throughput. The design space is generally very large, thus demanding an intelligent search method to arrive at a solution within a reasonable amount of time. Common techniques include integer linear programming, Markov decision processes, Pareto optimality, and dynamic programming, well as heuristic searches such as simulated annealing, genetic algorithms, tabu search, and design-space pruning. An FPGA design space can be searched at many levels, from the low-level specification of individual look-up tables to high-level complex modules. A sampling of some of the different techniques and applications for DSE are discussed in this section.

An overview of the different types of processors that are typically considered in a design space search is provided in [25]. Reduced Instruction Set (RISC), Complex Instruction Set (CISC), VLIW (Very Long Instruction Word), dataflow, tagged-token, and pipelined architectures are all commonly utilized. A design space explorer is generally restricted to one flavor of processor in order to put an upper bound on the time needed to search the design space. Trying to search across all possible architectures is considered to be an intractable problem.

Generally used heuristics for design space exploration include the comparable techniques of simulated annealing, genetic algorithms, and tabu search. A study has been performed which compares the three methods, arriving at the conclusion that tabu search may be better in some instances [26]. In [27], a good description of performing design space exploration for a reconfigurable processor is described. Important elements to be considered in the design space include allocation of computational, control, and memory resources, along with the scheduling of operations onto these resources. Exploration can occur in both parallelization (spatial optimization) and pipelining (temporal optimization). Simulated annealing is employed as the heuristic search method. Over thousands of iterations of the simulated annealing algorithm, the throughput of the processor gradually improves.

Additional tools that use simulated annealing coupled with the concept of Pareto-optimality are presented in [28] and [29]. A solution is said to be Pareto-optimal if there does not exist a solution that betters one parameter without worsening one or more of the others. These Pareto-optimal solutions can be used to guide the search of the simulated annealing algorithm. While a Pareto-optimal solution may not be a globally optimal solution, it is a good candidate for an area in which to focus the search. This technique was applied to a MIPS processor platform with adjustable data cache, instruction cache, and main memory sizes. Typical benchmarks in signal processing and image conversion were used to test the system.

In [30], a system has been developed to explore the space of heterogeneous micro-architecture processors, 20 to 50 of which may reside on a single FPGA. The search tool uses Integer Linear Programming as the search method, where the goal is to maximize throughput. Integer Linear Programming is a method for solving a system of linear inequalities. The linear constraints define a polyhedron, whose edges can be traversed until the optimal solution is found. Integer Linear Programming is an NP-complete algorithm that is often coupled with branch-and-bound techniques to reduce the compute time. The tool was tested by implementing an FPGA-based IPv4 packet forwarder that can outperform a hand-tuned design.

Another Integer Linear Programming DSE is presented in [31]. ILP is used to determine the most profitable extensions that should be added to a base processor for various data encryption and decryption techniques for high-bandwidth data. Extensions can include additional arithmetic units or combinations of units, such as a multiply-accumulate unit. The processor is targeted for implementation on an ASIC and the typical constrained optimization problem of throughput versus area utilization is solved.

The inventors of SUIF have also made significant inroads in the area of FPGA design-space exploration, specifically targeting source code that consists of multi-dimensional array accesses [32]. Concepts from SUIF have been combined with ideas from C-to-architecture tools to develop a method for exploring the time/space tradeoff in a custom FPGA architecture. The method involves the use of hardware synthesis tools from Xilinx or Altera as a mechanism for providing timing and area estimates for a potential design. The design is revised over many iterations of trial-and-error until an acceptable (but not necessarily optimal) architecture is discovered.

One technique for decreasing the time needed to find an architecture in the design space is a technique known as design space pruning [33]. Essentially, this method provides early estimates of area/latency tradeoffs to the search engine, immediately eliminating any solutions that are estimated to be poor performers by bounding the value of acceptable solutions. The searcher can then focus on optimizing more promising solutions. This technique can be successfully employed in combination with a standard exhaustive search, or with one of the other heuristic searches. Estimations are performed by modeling architecture efficiency based upon number of data-path operators, data bit-widths, register file size, number of control units, control signal complexity, total memory size, and number of read and write ports needed to support concurrent memory accesses. The resultant architecture would generally be a VLIW-style processor.

Another technique for DSE is to model the search problem as a Markov Decision Process (MDP) [34]. An MDP is a flavor of reinforcement learning in which a program traverses design states in a decision tree probabilistically according to values that have been learned over time. In other words, an MDP is initially a poor-performing random search. However, over thousands of trials, the MDP can be "trained" to produce high-quality architectures. This algorithm has been applied successfully to derive custom VLIW processors for various image and video compression algorithms.

When the design space consists of a traditional load/store processor that needs to be streamlined for a specific application, a technique such as CUSTARD [35] can be used to design an efficient multi-threaded processor. CUSTARD begins with a traditional MIPS integer pipeline processor. The processor is customized in three ways. First, unneeded instructions are removed from the instruction set. Second, additional pipelines are introduced to maximize parallelism. Each pipeline need not be identical; rather, each can be customized to support only the needed instructions. Third, the simple instructions supported by the MIPS RISC architecture can be combined to form more complex instructions (combining multiplication with addition to form a multiply-accumulate instruction is one common example). The cost of introducing complex instructions must be evaluated carefully, as they sacrifice versatility for speed. A cycle-accurate simulator is used to measure the performance of the candidate processors.

Use of a genetic algorithm is another popular method for design space exploration. For example, [36] details the use of a genetic algorithm for deriving a custom architecture for a digital camera. The processing platform consists of a PowerPC core, data cache,

instruction cache, memory, and buses. Different architectures are derived depending upon the file format and photo resolution.

In [37] an improvement to the basic genetic algorithm is proposed. A genetic algorithm maintains a population of current solutions at any given time during execution. A technique called "fuzzy clustering" is introduced which combines solutions into clusters or groups based upon score. The GA can then discard lower-scoring clusters and focus on the more-promising ones. This technique is once again applied to deriving VLIW architectures for video processing.

Finally, a project is presented in [38] in which an iterative improvement algorithm (based upon simulated annealing) is utilized to design processors for noise cancellation algorithms. The resulting processor is a VLIW processor with an arbitrary mix of multipliers, adders, and multiply-and-accumulate units.

In summary, DSE is a powerful tool. Many search strategies exist and many applications can be targeted. The tradeoffs between a heuristic approach such as simulated annealing (faster results) and an exhaustive approach such as integer linear programming (correct results) must be weighed carefully. DSE can take place at different design levels. Discrete components such as processors and memories can be combined in different ways, or the internals of the processor itself can be customized.

2.3.4 CASPER and ASPEN

ASPEN (Automated Scheduling and Planning Environment) [39] and CASPER (Continuous Activity Scheduling, Planning, Execution, and Replanning) [3] are tools that were developed at the Jet Propulsion Lab for use in modeling and implementing space-based mission planning and scheduling algorithms. ASPEN consists of a GUI-based design environment that supports a C-like programming language for modeling events that must be scheduled. CASPER is a stripped-down version of ASPEN that was designed to fly on the satellite, performing dynamic planning and continuous rescheduling of mission-critical events in real time. CASPER continuously runs an Iterative Repair algorithm to constantly improve and update the schedule. In traditional planning methods, events are labeled as either "in view" or "beyond the horizon". Only those events that fall within the event horizon are planned. Continuous planning takes a different approach. Rather than utilizing discrete event horizons, all tasks to be scheduled are available to the planner at all times. The tasks are divided into short-, medium-, and long-range types, where short-range tasks are the most detailed in terms of specific start-times, exact event durations, resource utilizations, etc., while long-range task representations are very general, rough estimates of timing and resources needs. This organization is shown in fig. 1.

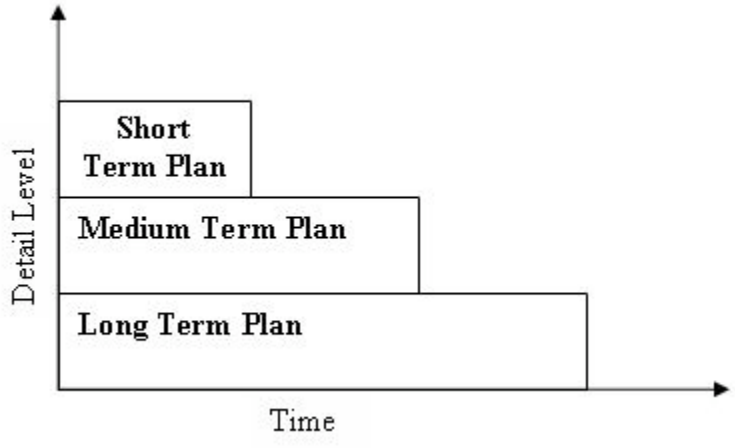


Fig. 1: Continuous planning partitions.

As time progresses, tasks pass from one planning frame to another, depending upon time left until execution. The planner runs constantly to improve the schedule. The Iterative Repair algorithm runs as detailed in fig. 2.

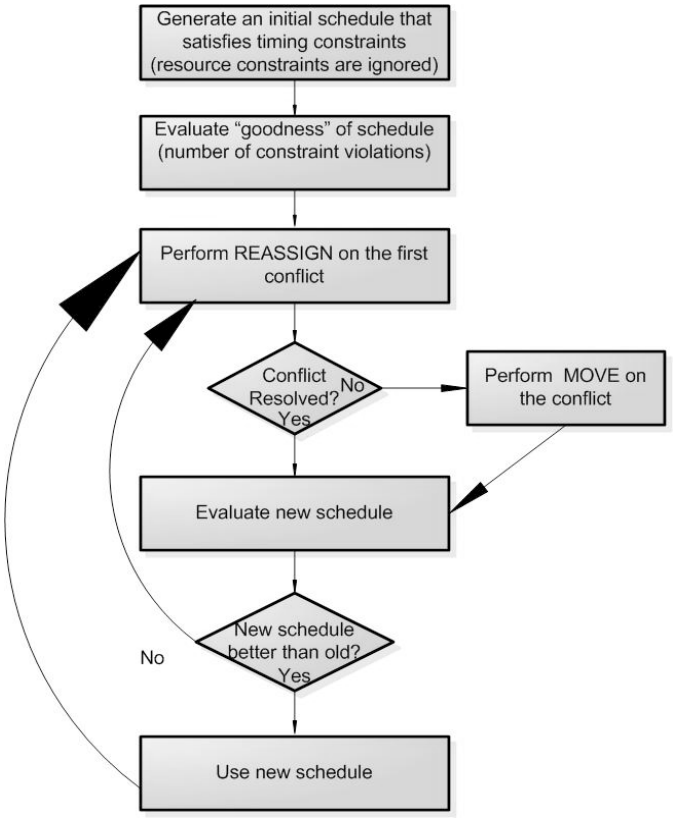


Fig. 2: Iterative Repair flowchart.

A first-guess solution is generated initially which satisfies timing constraints while ignoring resource constraints. This solution is then gradually improved, or repaired, over many iterations until an optimal (or close-to-optimal) schedule has been found. These

improvements are made by either reassigning an event to an unused resource or moving the event to a different time slot.

2.3.5 Heuristic Search Techniques

Iterative Repair is, by nature, a greedy algorithm. Schedules can be improved to a point, but the algorithm can become trapped in a local optimum in the search space. One or more inferior solutions may need to be used as stepping stones to arrive at the global optimum. This technique of sacrificing local optimally for the global good is employed by Simulated Annealing. Simulated Annealing is a method for performing combinatorial optimization on a large search space where an exhaustive search is not tractable. Simulated Annealing is based upon the metallurgic phenomenon of annealing, in which a metal is heated to an extremely high temperature and then allowed to cool slowly, resulting in a (near) minimal energy configuration of the crystalline structure. Simulated Annealing takes an initial solution and minimizes the "energy" of the solution through successive steps, in each of which a slightly different solution is compared to the current solution. In order to avoid becoming trapped in local optima, solutions with higher energies are accepted probabilistically, based upon the current temperature of the system. The temperature is gradually lowered over many iterations until an optimal solution is found.

As discussed above, the Iterative Repair algorithm for event scheduling utilizes the Simulated Annealing heuristic to effectively search for optimal solutions. Other common heuristic searches which yield similar results and which could be used in place of Simulated Annealing include Genetic Algorithms [40] and Stochastic Beam Search. A summary of all three methods is provided in [41]. Pseudocode for the three algorithms is presented below in figs. 3-5.

Simulated Annealing

```
generate initial solution
evaluate initial solution
set initial temperature (T)
Loop
      generate new solution
      evaluate new solution
      If new better than old
            overwrite old solution with new
      Else
            compute ΔE (ΔE = |old score - new score|)
            compute acceptance probability (p = exp(-ΔE/T))
            generate random number between 0 and 1
            If random <= acceptance probability
                  overwrite old solution with new
            EndIf
      EndIf
      lower T
EndLoop when T ≈ 0
```

Fig. 3: The Simulated Annealing algorithm.

Stochastic Beam Search

```
generate k initial solutions
evaluate k initial solutions
Loop
      generate k solution
      evaluate k solution
      merge old and new solutions for 2k solutions
      select k best solutions
EndLoop when improvement ceases
```

Fig. 4: The Stochastic Beam Search.

Genetic Algorithm

```
generate k initial solutions
evaluate k initial solutions
Loop
      Loop k times
            select random solutions x and y from k
            crossover x and y to form new solution
            generate random number between 0 and 1
            If random <= mutation probability
                  mutate new solution
            EndIf
            evaluate new solution
            add new solution to pool
      EndLoop
      select k best solutions out of 2k available
EndLoop when improvement ceases
```

Fig. 5: The Genetic Algorithm.

All three algorithms operate on variations of the same general premise: gradually improve a solution (or set of solutions) over time until a solution of sufficient quality is discovered. Simulated Annealing (SA) operates on a single solution, while Stochastic Beam Search (SBS) and Genetic Algorithm (GA) maintain a pool, or population, of solutions (solutions are also termed chromosomes in GA). All three algorithms utilize similar subroutines for evaluating solutions and generating new solutions, although the crossover and mutation operators used in GA are a bit more complicated. All three algorithms also depend heavily on random number generation and probabilities. GA and SBS differ from SA in the technique used for avoiding entrapment in local optima. In SA, suboptimal moves are accepted as long as the temperature is sufficiently high and the value of the new solution is reasonably good. GA and SBS solve the problem by maintaining a pool of current solutions such that it is statistically impossible for all solutions to fall in an area around the same local optima.

In theory, implementing combinatorial search algorithms in hardware could significantly speed-up the search process. Large amounts of parallelism and pipelining can be extracted from SA, GA and SBS, since deriving a new generation is largely only a function of the previous generation. Hardware-based GA implementations abound in the literature. Some recent examples of FPGA-based GAs are discussed here.

A GA has been implemented on an FPGA for the purpose of blind signal separation [42]. Function-level pipelining has been implemented. The high-level flow chart, which is almost identical for all hardware GAs surveyed, is shown in fig. 6. This system was implemented on a Xilinx Virtex FPGA, using a chromosome length of 64 bits and a

population size of 80 chromosomes. As long as the chromosome length is kept reasonably small, this type of technique in which entire chromosomes are passed between pipelined modules works well.

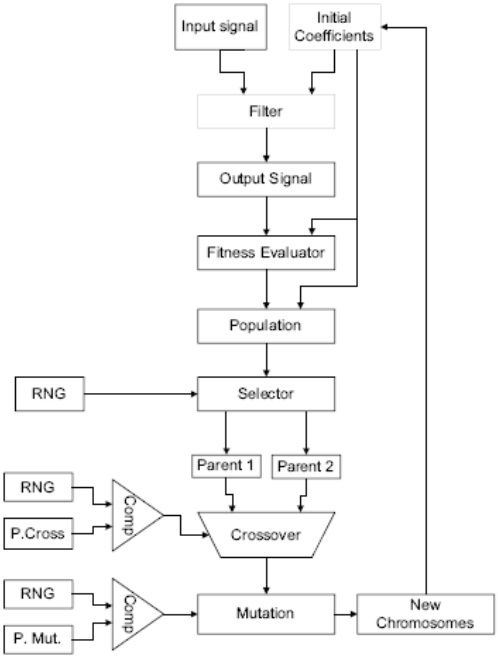


Fig. 6: Hardware implementation of a Genetic Algorithm with short chromosomes [42].

GAs are also commonly used for filter design. In [43], a GA for grey-scale filter design has been implemented on a Xilinx Spartan II FPGA. A pipelined approach similar to the one shown in fig. 6 is once again taken. Chromosomes are 16 bits in length and a generation consists of 10 chromosomes. Another example is using a GA to perform function interpolation [44]. In this case, chromosomes are 24 bits in length or less. A system for maximizing algebraic expressions has been developed with 5-bit chromosomes [45]. Three different sets of libraries and design algorithms written in both Verilog and VHDL for the implementation of various types of GAs with relatively small chromosome lengths have also been developed [46-48].

The examples discussed above are very simple, as the chromosome length in all cases is so small that the entire chromosome can be passed between pipelined modules simultaneously. Most real-world problems, however, are complex enough that a chromosome can be hundreds of bytes in length. The simple pipelined implementations discussed above clearly cannot handle this complexity, as 100-byte wide busses are not feasible for implementation on an FPGA. One method for resolving this problem is to split the chromosome into manageable chunks and transfer one chunk of data on each clock cycle [49]. This allows a modified version of the simple pipelined architecture to be created. Obviously, as the ratio between chromosome length and bus width grows, the time used in data transfers also increases.

Traveling Salesperson is the classic combinatorial search problem, the goal being to find shortest path for visiting every node on a graph exactly once. A version of the problem has been solved using GA on a Xilinx Virtex-E FPGA [50]. Rather than coding

modularly in VHDL or Verilog, the implementation was performed using Handel-C. Population size is 200 chromosomes, with the chromosome size being variable. Explicit pipelining is not realized in this design, as Handel-C is a behavioral language rather than a structural language. Any parallelization is specified by the programmer and interpreted by the Handel-C compiler.

An example of performing SBS in hardware was used for speech recognition algorithms [51]. Chromosome widths in this example are once again limited to a few bits. Because of the inherently serial nature of the SA algorithm, it is not generally considered an interesting problem for parallelization or hardware implementation.

The chromosomes needed for scheduling are much larger than what can be transmitted on a bus in one shot. For example, take a system in which 100 tasks need to be scheduled within a 24 hour period. If the tasks need to be scheduled with a resolution of 1 minute, the chromosome would consist of 100 11-bit numbers. An 1100-bit chromosome is much too large to transfer as a single chunk. Novel architectures must be developed to accelerate this algorithm.

A hardware implementation of the FPGA place-and-route algorithm using Simulated Annealing has been done [52]. The place-and route algorithm searches for a layout of a circuit on an FPGA, including the use of both logic blocks and routing resources, that minimizes resource utilization while maximizing circuit performance. The architecture in this case is based upon systolic arrays, rather than the pipelined structure advocated in this paper. Comparable speedups approaching three orders of magnitude were found using this architecture.

```
temperature ← INITIAL_TEMP
generate initial solution
compute score of initial solution
while temperature > STOP_THRESHOLD
     copy: next_solution ← current_solution
     alter: modify next_solution
     evaluate: compute score of next_solution
     accept: probabilistically accept next_solution
     adjustTemperature
```

Figure 7: Simulated annealing pseudocode. An optimal solution is derived by repeatedly executing the five steps.

As Simulated Annealing is the heuristic search employed by Iterative Repair, a detailed description of the algorithm is now provided. As described in fig. 3 and revised in fig. 7, an initial solution is generated, usually randomly, and evaluated. This initial solution is designated as the current solution until a new one is accepted. The main loop is now entered, which generally loops several thousand times. On each iteration, the current solution is copied verbatim to a second buffer, where it is designated as the next solution. This next solution is then altered slightly and evaluated. The score of this new solution is then compared against the score of the current solution. The crux of the algorithm is determining whether to accept the next solution as the new current solution or discard it in favor of the keeping the resident current solution. This decision is made according to (1).

$$p = e^{\Delta E/T},\ \Delta E = S_{next} - S_{current} \quad (1)$$

In this equation, S_{next} and $S_{current}$ are the scores of the current and next solutions, respectively, and T represents temperature. The probability p is a function of both the temperature and the difference between the score of the current solution and the score of the new solution (ΔE). A random number is generated and compared to p to determine whether a solution should be accepted. When the temperature is high, suboptimal solutions are more-likely to be accepted. This feature allows the algorithm to escape from local minima as it searches the solution space and zero in on the true optimal solution. The last step in the loop decreases the temperature according to a pre-determined schedule. A typical method is to geometrically decrease the temperature by multiplication by a cooling rate, which is generally a number such as 0.99 or 0.999. The closer the cooling rate is to 1.0, the more times the loop will execute. This results in longer program execution, but also improves the probability of finding the best solution. Cooling too fast reintroduces the local-optima entrapment problem to the system.

Fig. 8 shows how simulated annealing can be applied to an iterative repair problem. In this case, a simple example consisting of ten events is presented. These events are numbered 0 through 9, and can be treated as indices to a solution array in computer memory. At any point in time, two solutions are maintained: the current solution and the potential next solution. From fig. 3, the first step in the loop is to copy the current solution into the next solution buffer. Once this is done, the next solution must be altered in some way. Fig. 8 depicts a simple value swap, where two events are selected at random and the respective start times are swapped.

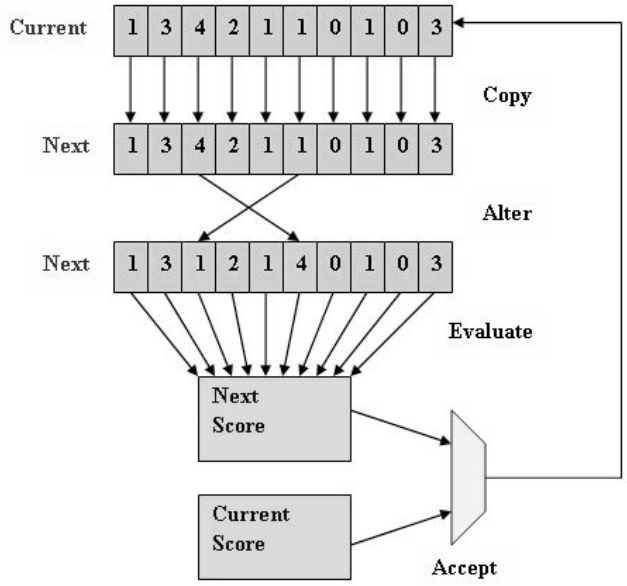


Figure 8: An example of iterative repair using simulated annealing. A solution is copied, altered, evaluated, compared against the current solution, and accepted conditionally. This process is repeated thousands of times to arrive at the optimal solution.

This new solution must then be evaluated to determine how it compares with the current solution. Factors to take into consideration when computing the value of a schedule could include effective resource utilization, dependency graph violations (when a child event is scheduled before a parent event), and overall length of the schedule. Once the new schedule's value has been determined, equation (1) is applied to determine whether or not it should replace the current solution. If the next score is better than the current score, it replaces the current score unconditionally. If it is worse, it is accepted with the computed probability, depending on both the score of the solution and the temperature. The temperature is then updated as described previously. This process is repeated until the temperature falls below a predetermined threshold, at which time the best schedule found is returned by the system.

2.3.6 Application-Specific Instruction Processors

Application-Specific Instruction Processors (ASIPs) are a relatively new method of hardware architecture design. When the target application is known at hardware design time, the processing platform can be customized to execute the application in an efficient manner, rather than using generic, general-purpose processors. In [53] some of the benefits of ASIPs over standard processors, including execution time and power consumption are explained. The ASIP approach is very compatible with FPGA technology. A single FPGA can be repeatedly reconfigured to implement an ASIP for a variety of applications. As FPGA technology improves in terms of clock speeds and power consumption, ASIPs gain popularity as an alternative to traditional ASIC design methodologies [54]. In this section, techniques for ASIP design, implementation, and optimization are discussed.

One technique for implementing an ASIP is to combine a conventional microprocessor with a reconfigurable module [55]. For a specific application, the reconfigurable module is programmed to perform one or more complex, application-specific instructions more efficiently than possible in the microprocessor itself. In other words, an FPGA serves as a coprocessor to accelerate computations. Determining which sets of instructions should be combined and placed on the FPGA is a challenging task. A smaller hardware module that can be used multiple times may prove more efficient than a larger module that can only be used once or twice.

Another method for ASIP implementation is to design a custom Very Long Instruction Word (VLIW) processor. A VLIW processor consists of multiple computational blocks that are coupled with a wide instruction word, allowing for multiple instructions to be issued simultaneously. A VLIW processor is an excellent choice for exploiting instruction-level parallelism. In [56], a VLIW-based technique termed a Wide Counter-flow Pipeline is proposed. The architecture consists of a pipelined VLIW processor, with instruction width and computational blocks dictated by the application, coupled with a counter-flow pipeline that brings results back to the register file. A VLIW-style processor efficiently manages compute-intensive code with minimal control flow. Little instruction-level parallelism can be extracted from branch-heavy code.

In [57] the steps necessary to derive an efficient ASIP are identified. First, application behavior must be analyzed utilizing specialized compilers [58]. Second, a base architecture must be selected. Possible architectures include VLIWs, specialized co-processors, data-flow processors, etc. Once the architecture has been selected, optimal speedups must be determined, such as VLIW widths, co-processor latencies, register file [59] and memory sizes [60], etc. Exploring this design space is a combinatorial search problem, as there are generally thousands or millions of combinations of possible architecture details that need to be explored. Examples of heuristic techniques for searching this design space for a VLIW processor are detailed in [61] and [62]. Finally, a tool must be developed to both generate a Hardware Description Language (HDL) representation of the architecture and then map the source code onto the novel processor. Schliebusch et al. [63] describe a method for implementing this tool by introducing a hardware intermediate format.

2.3.7 Using FPGAs in the Space Enviroment

The space environment is not electronics-friendly. The sun is constantly spewing large amounts of fast-traveling, highly-charged particles into space in a phenomenon known as the solar wind [64]. These high-energy particles can affect electronic circuits in a variety of ways, causing single-event upsets (SEUs) and single-event latchups (SELs). SEUs occur when a transistor is energized by a high-energy particle, resulting in a bit flip. This phenomenon can occur in any part of a circuit, resulting in temporary data corruption, code corruption, or, in the case of an FPGA, even hardware architecture corruption. A SEL is similar to a SEU, occurring when a high-energy particle permanently damages a transistor rendering it unusable. Techniques have been developed for implementing fault-tolerant circuits on FPGAs. In [65] a summary is provided of current techniques for implementing fault tolerance on SRAM-based FPGAs.

For example, in [66], a system is specified for utilizing Triple Modular Redundancy to provide fault tolerance against SEUs. In this scheme, the circuit is duplicated on three different FPGAs. A voter mechanism is employed to produce the result, assuming at least two out of the three FPGAs are operating correctly. An external microcontroller and radiation-hardened PROM complete the circuit. When voting is not unanimous, the microcontroller reprograms the faulty FPGA using bit-stream data read from the PROM. A similar paradigm, utilizing radiation-hardened FPGAs as both controllers and processors, is presented in [67]. Another work utilizing a radiation-hardened ASIC as the voter is described in [68].

Techniques have also been developed for recovering from SELs, which permanently damage the FPGA fabric. In this case, portions of a circuit residing in a damaged sector must be moved to a physically close intact and unused region. In [69] evolutionary techniques are used to determine how hardware blocks should be placed on an FPGA to facilitate optimal rearrangement. At design time, a genetic algorithm is used to determine the most flexible manner in which a circuit can be mapped onto an FPGA, creating simulated faults to observe needed patterns of rearranging.

3. HARDWARE TEMPLATE

Before an algorithm can be presented for deriving Iterative Repair processors, an architecture template is described. Any simulated annealing search problem can be mapped on to this template. The template described in this section is generalized from a from a specific ANSI C implementation of Iterative Repair using simulated annealing. In this implementation, a solution is represented as a string of start times for events numbered 0 to 99 for a problem consisting of 100 events that need to be scheduled. Events have dependencies, meaning that certain events must complete before others can start. Fig. 3.1 depicts this event dependency graph. Each event utilizes one unit of one of four types of resources. There are four resources of each type.

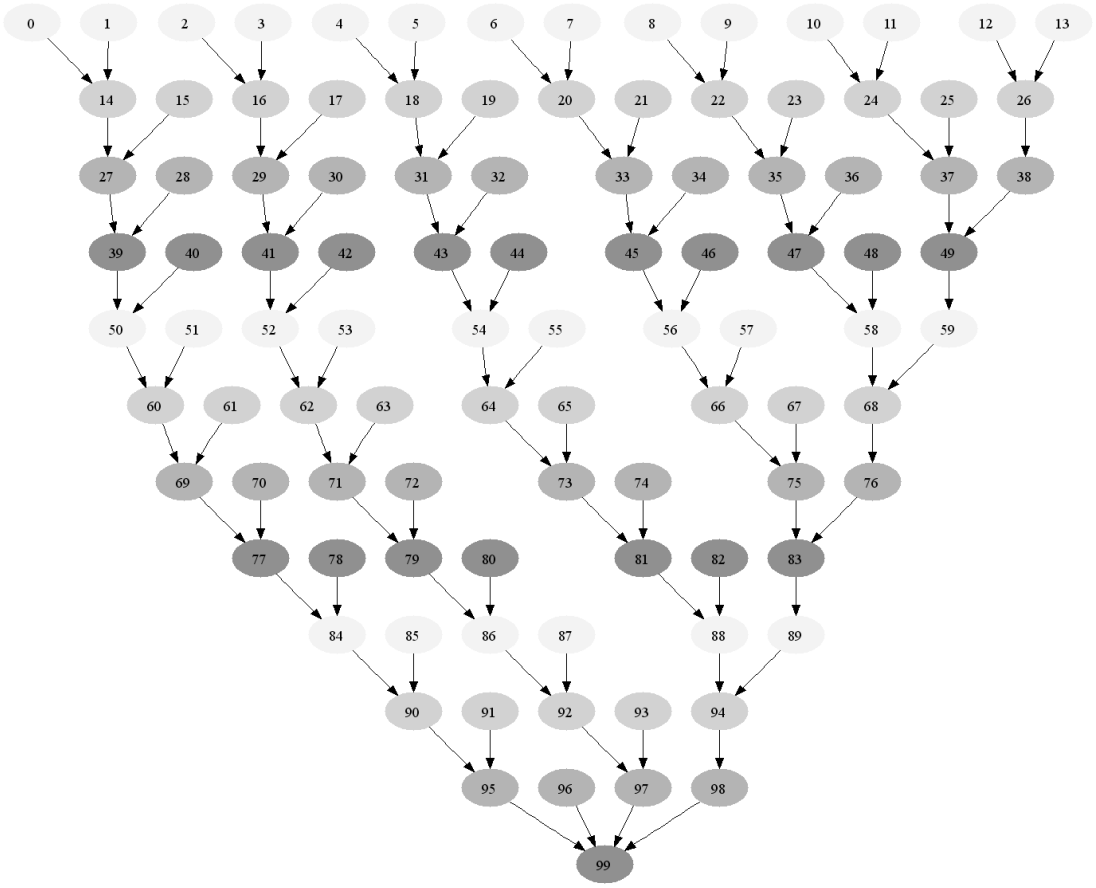


Figure 9: Event dependency graph of 100 events. Events are represented by the numbered nodes. Edges indicate dependencies. Each event also uses one of four resource types, designated by the shape of the node.

The resource type associated with each event is designated by the shape of the event node in fig. 9. Each event takes one time step to complete. Additional input parameters are a maximum schedule length of 32 time steps, an initial temperature of 10,000, a cooling rate of 0.9999, and a termination threshold of 0.0001. This means that the schedule cannot exceed 32 time steps, the simulated annealing temperature starts at 10,000 and is decreased geometrically by 0.9999 on each iteration, and the program terminates when

the temperature falls below 0.0001. This means that the loop runs 184,198 times. For this design, 16-bit integer arithmetic and 32-bit floating-point arithmetic were assumed. Based upon the pseudocode described in section 2, an application-specific architecture was developed to exploit the characteristics of the algorithm. The architecture is composed of a four-stage pipeline coupled with five memory banks. Each stage in the pipeline corresponds to a step in the simulated annealing pseudocode – copy, alter, evaluate, and accept. A global controller coordinates execution and data exchange between the units. An interface between memory banks and processors is provided. An Adjust Temperature Processor controls the cooling process. As this is a pipelined architecture, it can only operate as fast as the slowest stage. Careful design techniques must be employed in the more complex stages to minimize the latency. A block-level diagram of this architecture is shown in fig 10. Each of these stages is discussed in detail in this section.

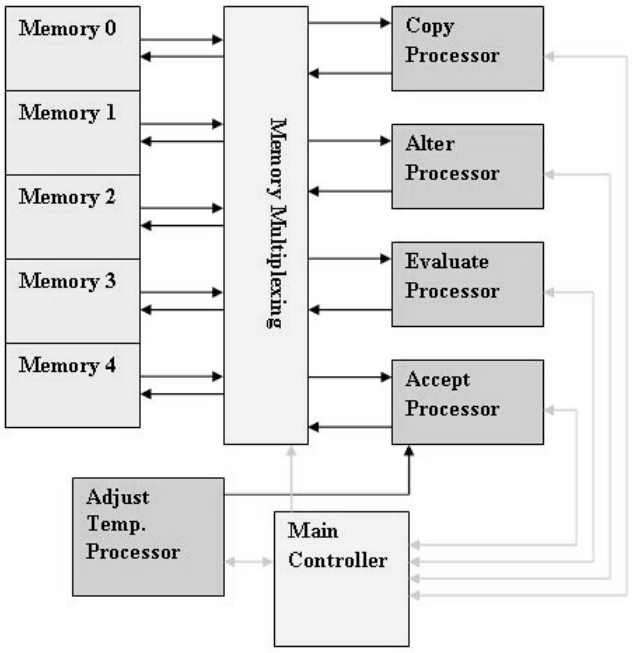


Figure 10: Iterative repair architecture. A pipelined processor template with associated memory constructs is derived from the simulated annealing pseudocode.

3.1 Memory Design

The architecture consists of five memory banks, numbered zero through four in fig. 10, derived from Xilinx FPGA block RAMs. Each memory bank needs one write port and four read ports, all 16 bits wide. Four read ports are needed to facilitate parallelism in the Evaluate Stage. Because a Xilinx block RAM allows for one read and one write port, four block RAMs are used in the instantiation of each 128-word (16-bit word) memory bank. Each memory bank holds a solution and the score of the solution. The memory contents are detailed in fig. 11. At any given point in execution time, one memory bank is associated with each of the four processing stages in the pipeline. The remaining memory block holds the current solution. The main controller determines how memory

blocks are associated with different processing stages. Details on the manner in which memory banks are managed are discussed in the section on the main controller. Because the location of data in the memory is known at design time, many cycles are saved during execution by avoiding address computations.

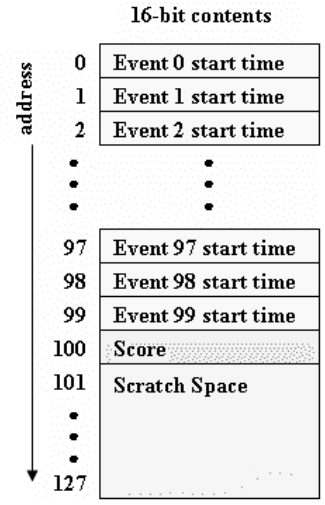


Figure 11: Memory contents. Each of the five memory modules is configured identically.

3.2 Copy Processor

As shown in fig. 7, the main loop of the simulated annealing algorithm begins by making a copy of the current solution. This copy is then altered to generate a new solution that could potentially replace the current solution. In the architecture shown in fig. 10, the Copy Processor performs this copying function. The C code for performing the copy function is shown here:

```
for (i=0; i<MAX_EVENTS; i++)
        dest[i] = source[i];
```

Since the length of the solution is known; the contents of the solution in the "current solution" memory bank are copied, word by word, into the memory bank currently associated with the Copy Processor. There is no need to accelerate the copy process through parallelism, as this pipeline stage is guaranteed to complete in $n+1$ clock cycles for a solution length of n. Other stages are much more compute-intensive. The copy processor is simply a controller to facilitate data transfers. A "step" signal comes from the main controller, indicating that a new pipeline step has begun. The copy controller consists of a counter that generates addresses and produces a "done" signal when all data has been copied and also controls the write-enable line on the destination memory bank. The source and destination addresses are identical, because the data locations in each memory bank are identical, as shown in fig. 11.

3.3 Alter Processor

The second stage in the iterative repair pipeline is the Alter Processor. One event is selected at random from the solution string. The start time of this event is changed to a random time that falls between zero and the maximum latency. The C code for this function is as follows:

```
i = rand() % MAX_EVENTS;
j = rand() % MAX_LATENCY;
sched[i] = j;
```

The hardware implementation of this stage, shown in fig. 12, could be accelerated by introducing an additional random number generator and an additional divider, allowing for maximum concurrency. This additional hardware is not necessary however, as a 15-cycle integer divider allows this stage to terminate in 21 clock cycles, regardless of the size of the solution string. As solutions generally consist of hundreds of events, even the simple Copy Processor will have a greater latency than the Alter Processor. The alter controller is based on a counter that starts when the "step" signal is received from the Main Controller, control logic to enable register writing on the "address" and "data" registers on the proper clock cycles, and a "done" signal.

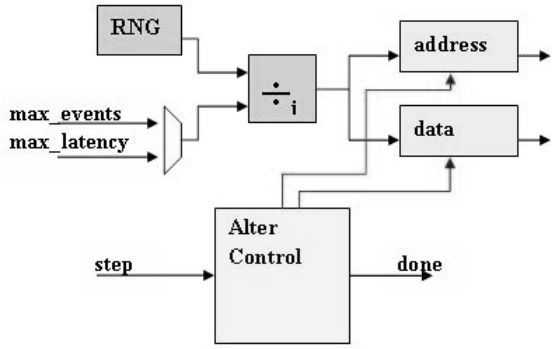


Figure 12: The Alter Processor. A random number generator is used to select and modify the start time of one event.

The random number generators (RNG) used in both the Alter Processor and the Accept Processor are 15-bit linear feedback shift registers (LFSRs) which generate a new integer between zero and 32,767 on every clock cycle. Bits 14 and 13 of the shift register are tapped and passed through an exclusive-or gate to derive the incoming bit.
Some improvements could be made to the alter processor to further enhance performance in both time and resource utilization. For example, if "MAX_LATENCY" and "MAX_EVENTS" were constrained to be powers of two, the integer divider in fig. 12 could be replaced with a simple shift register. The feasibility of constraining "MAX_EVENTS" to be a power of two, however, is low for most scheduling problems, as the number of events is rarely if ever a perfect power of two. Leaving the divider unit as is allows the architecture to handle problems of varying sizes.

3.4 Evaluate Processor

The Evaluate Processor is by far the most complex of all the pipeline stages in the iterative repair architecture. The task of this processor is to compute a numerical score for a potential solution. Manual design space exploration was performed to arrive at an optimal architecture. The score of a solution to this particular iterative repair problem consists of three components. A penalty is incurred for total clock cycles consumed by the schedule. A second penalty is assessed for double-booking a resource on a given clock cycle. Thirdly, a penalty is assigned for dependency violations, which occur when event "b" depends upon the results of event "a", but event "b" is scheduled before event "a". The partial scores from each of these three components are weighted and summed to produce the solution score.

As part of the simple design space exploration used to design this processor, the entire evaluate stage was initially designed as a sequential processor, which resulted in a stage latency of over 600 clock cycles. Because of this latency, it was elected to exploit the parallelism inherent to the algorithm. Each of the three evaluation components described above is implemented as an individual pipelined processor. Because the three components of the score can be computed independently and combined at the end, all three processors can run in parallel, thus saving substantial clock cycles. The first sub-processor, termed the Dependency Graph Violation Processor, or DGVP, is shown in fig. 13.

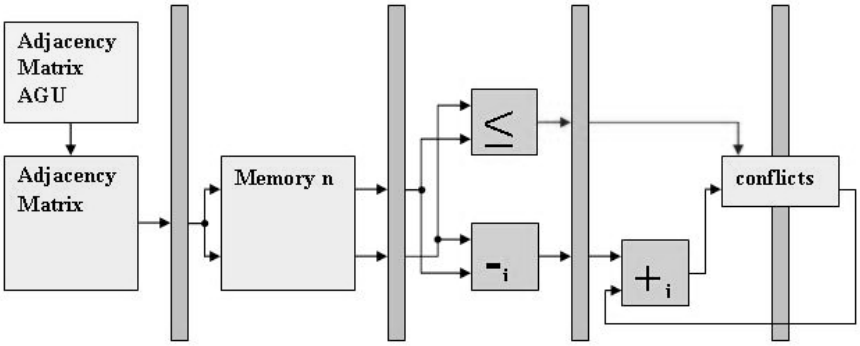


Figure 13: Dependency Graph Violation Processor architecture. This four-stage pipelined processor computes all dependency graph violations for a given schedule.

The original C code from which this processor is derived is shown here:

```
for (i=0; i<MAX_EVENTS; i++)
        for (j=0; j<MAX_EVENTS; j++)
                if ((a_matrix[i][j] == 1) && (sched[j] <= sched[i]))
                        conflicts = conflicts + (sched[i] - sched[j]) + 1;
```

The processor is a four-stage pipeline. In the first and second stages, an adjacency matrix is used to index the solution memory and determine when parent/child pairs of events are scheduled. A naïve approach to both software and hardware design would be to implement the adjacency matrix with a location for every parent/child combination. A one is placed in the matrix when a connection exists. For example, in fig. 9 there is a

connection from event one to event 14. This is represented by placing a one in a_matrix[1][14]. A more efficient method for representing the adjacency matrix, especially for sparsely-connected graphs, is to keep track of only those connections that actually exist. While looking through the traditional matrix would incur wasted cycles on all event pairs that weren't adjacent, providing a list of adjacent pairs removes the "if" construct from the C code and reduces the complexity of the stage. In the example in question, this reduces the number of look-ups needed from 10,000 (all combinations of 100 source events and 100 destination events) to only 99, which is the actual number of dependencies in fig. 9.

The third and fourth stages determine the magnitude of the penalty, if any, to be incurred because the child event is scheduled before the parent event terminates. The magnitude of the penalty encourages offending parent/child pairs to gradually move toward each other, thus decreasing the penalty over several iterations and causing the schedule to become more optimized.

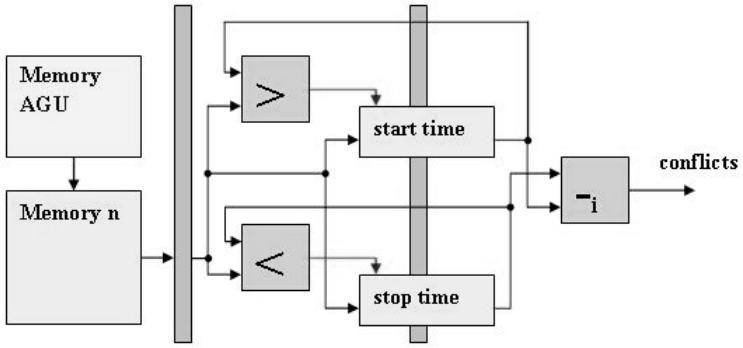


Figure 14: The Total Schedule Length Processor architecture.

The second sub-processor, shown in fig. 14, is the Total Schedule Length Processor (TSLP). Its job is to simply compute the total length of the schedule from beginning to end. This 2-stage processor looks through all events one-by-one, updating the earliest and latest times seen so far. Upon conclusion, the difference between the earliest and latest times is the schedule length. The C code for this process is shown here:

```
for (i=0; i<MAX_EVENTS; i++) {
        if (sched[i] < start)
                start = sched[i];
        if (sched[i] > stop)
                stop = sched[i];
}
conflicts = stop - start;
```

The third sub-processor internal to the Evaluate Processor is the Resource Over-Utilization Processor (ROP). This processor, depicted in fig. 15, is responsible for checking for resource over-utilization on every resource for every time step.

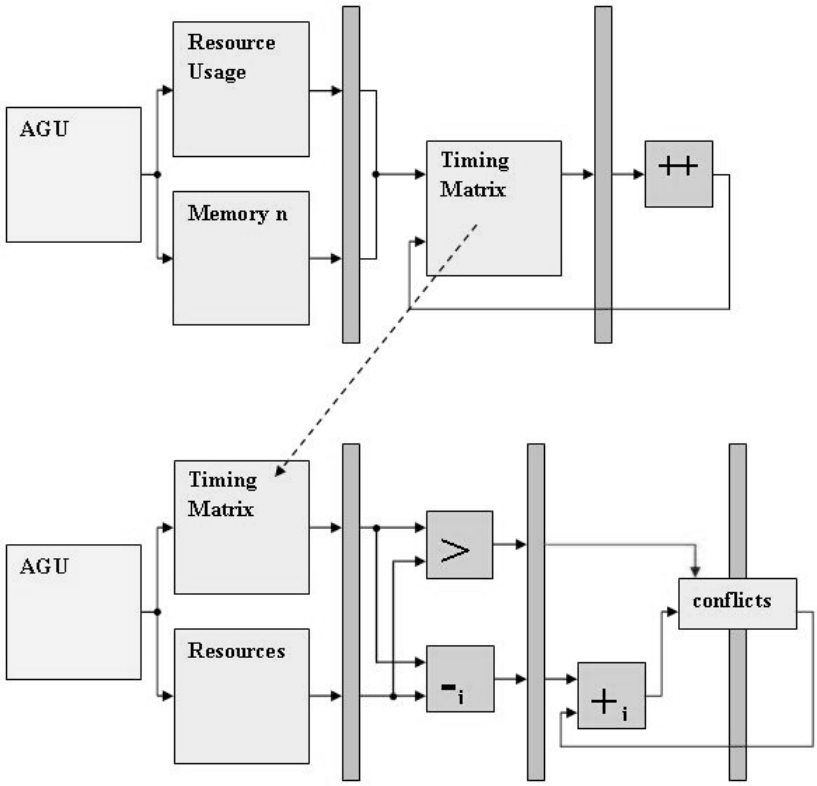


Figure 15: The Resource Over-utilization Processor architecture. A timing matrix is first populated and then compared against the available resources.

This processor is actually two different pipelined processors. The first populates a timing matrix, which is a two-dimensional matrix that keeps track of the resource utilization of every resource for every time step. This matrix is populated by going through the events one by one and determining when each is scheduled and what resource each uses. The C code for this process is shown here:

```
for (i=0; i<MAX_EVENTS; i++)
        t_matrix[sched[i]][res_usage[i]]++;
```

This timing matrix is then passed on to the second processor, in which the utilization of each resource at each time step is compared to the total number of available resources of that type. When over-usage occurs, the amount of over-usage is added to the existing penalty. The C code for this is shown here:

```
for (i=0; i<MAX_LATENCY; i++)
        for (j=0; j<MAX_RESOURCE_TYPES; j++)
                if (t_matrix[i][j] > resources[j])
                        conflicts = conflicts + (t_matrix[i][j]
                        resources[j]);
```

All three sub-processors have “done” signals. When all three have completed their tasks, the three penalty values are summed to give the total score for the given schedule of

events. This score is stored in the associated main memory bank as depicted in fig. 11. The Timing Matrix must be cleared on each iteration. To avoid using clock cycles on this clearing operation, the Timing Matrix is implemented as a double memory (sometimes called a ping-pong buffer). On a given iteration, one block is used for computations while the other is being cleared. Unlike the other stages in the pipelined architecture which remain static for any iterative repair problem, the size and speed of the Evaluate Processor are not fixed, but rather are dependent upon the size and complexity of the list of events to be scheduled.

It should be noted that the different aspects of the solution score may have different significance. For example, dependency graph violations and resource over-utilization problems are much more severe than the total schedule length. The partial scores from the DGVP and ROP can then be weighted more-heavily than the score from the TSLP in the final score. This weighting can be done with minimal timing implications by performing a one-bit left shift operation to the DGV and ROP scores and a one-bit right shift to the TSLP score, thus effectively giving the DGV and ROP scores four times the weight of the TSLP score. This ensures that the critical violations are given first priority for removal, while TSLP scores are a secondary consideration.

3.5 Accept Processor

The Accept Processor's job is to determine whether to accept the next solution as the new current solution. If the next solution is better than the current solution, the next solution is accepted unconditionally. A solution that is worse than the current solution can also be accepted with a computed probability, defined in equation (1). The C code for this process is shown below:

```
delta_e = cur_value – next_value;
p = exp(((float)delta_e)/temperature);
if ((rand() / (float) RAND_MAX) < p) {
        for (i=0; i<MAX_EVENTS; i++)
                schedule[i] = next_schedule[i];
        cur_value = next_value;
}
```

An architecture that supports this computation is shown in fig 16. This processor mixes floating-point numbers with integer numbers, thus necessitating the integer-to-float conversion module shown. The "current score" and "next score" parameters are integers, while the "temperature" and "rand_max" parameters are 32-bit floating point. The current score and the next score are read from their respective memory banks. The temperature is provided by the Main Controller. The random number generator (RNG) is a 15-bit tapped shift register, described previously. The RNG and divider are used to generate a number between zero and one that is compared against the acceptance probability (p) to determine whether or not the new solution should be accepted. The exponential block is a floating-point unit consisting of a BRAM-based lookup table containing 1000 entries representing floating-point input values ranging from negative

infinity to zero and floating-point output values ranging from zero to one. The I-to-F block is an integer to single-precision floating-point converter. Notice that the computed acceptance probability is a function of both the quality of the score and current temperature. As the temperature decreases, suboptimal scores are less-likely to be accepted.

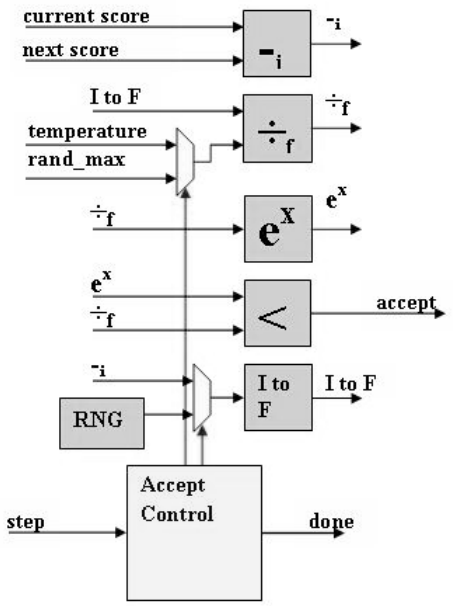


Figure 16: The Accept Processor. The new solution is always accepted if it is better. If worse, it is accepted with a computed probability.

3.6 Adjust Temperature Processor

The Adjust Temperature Processor is a simple but critical stage in the pipelined processor. The temperature is used to compute the probability of acceptance in the Accept Processor and by the Main Controller to determine when the algorithm should complete. There are many options for implementing a cooling schedule for a simulated annealing problem. In this case a popular geometric cooling rate was used, although this could easily be replaced with a different function deemed more appropriate for a specific application. The architecture for the Adjust Temperature Processor is shown in fig. 17. The current temperature is stored in a register. When the "step" signal is received, the temperature is multiplied by the constant "cooling rate", which for this architecture is set to 0.9999. This cooling rate allows the temperature to decrease slowly and geometrically, allowing for the discovery of better solutions.

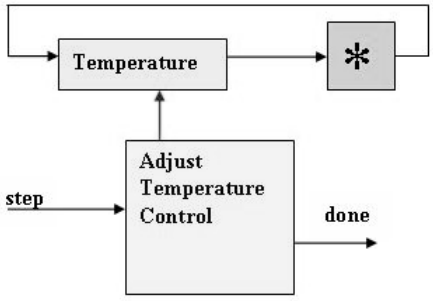


Figure 17: Adjust Temperature Processor. The temperature is reduced geometrically each time this processing stage runs.

3.7 Main Controller

The main controller is responsible for coordinating the sharing of data between processing stages, for allowing the pipeline to step ahead at appropriate times, and for determining when execution is complete.

The main controller coordinates the sharing of data between stages by keeping track of the memory block that is associated with each processing stage. Upon the completion of a pipeline period, the main controller must determine how to reassign the memory blocks to the different stages, keeping track of which one holds the current solution and which one can be recycled and assigned to the Copy Processor. This decision process is detailed in fig. 18.

a

Copy Mem	→ Alter Mem
Alter Mem	→ Eval Mem
Eval Mem	→ Accept Mem
Accept Mem	→ Copy Mem
Current Mem	→ Current Mem

b

Copy Mem	→ Alter Mem
Alter Mem	→ Eval Mem
Eval Mem	→ Accept Mem
Accept Mem	→ Current Mem
Current Mem	→ Copy Mem

Figure 18: Method for passing memory block pointers between processing stages when (a) the solution in the Accept stage is NOT be accepted and (b) the Accept stage solution is accepted.

Two different patterns of moving memory block pointers between processors are needed. One for when the solution associated with the Accept stage should be accepted as the new current solution and another pattern when the solution associated with the Accept stage should simply be thrown out. In the case where the new solution should not be accepted, shown in fig. 18a, the Current Pointer is not updated and all other pointers are passed to the next stage. In the case where the solution should be accepted, shown in fig. 18b, the memory pointer from the Accept stage is transferred to become the Current Pointer and all other pointers are passed to the next stage. The main controller also performs global synchronization. As shown in fig 10, the main controller receives a "done" signal from each of the stages. When all stages have completed, the main

controller sends out a "step" signal to each processor, indicating that they can proceed. The main controller also monitors the temperature and halts the system when execution is complete.

3.8 Template Performance

The FPGA resources needed to solve this scheduling problem are shown in Table 1.

Table 1: Resource consumption and speed

	Slice Count	DSP48 Units	BRAM	Latency	Max. Freq. (MHz)
Main Controller	160	0	0	3	472
Copy Processor	30	0	0	101	289
Alter Processor	390	0	0	21	232
Eval. Processor	317	0	5	233	222
DGVP	94	0	1	102	347
TSLP	64	0	0	101	307
ROP	164	0	4	232	310
Accept Processor	1,408	0	1	54	197
Adjust Temperature Processor	173	5	0	12	444
Memory	966	0	20	N/A	912
Complete Processor	**2,831**	**5**	**26**	**235**	**197**

Each of the five memory banks uses 4 BRAM blocks, thus the 20 blocks used by the Memory Module. The problem contains 99 dependency edges. The Dependency Graph Violation Processor (DGVP) in the Evaluate Processor needs to look at all 99 edges, plus three cycles for the pipeline delay, giving a total of 102 cycles. The Total Schedule Length Processor (TSLP) needs to look at all 100 events, plus one cycle for pipeline delays, yielding 101 cycles. The Resource Over-utilization Processor (ROP) needs to look at every event to populate the Timing Matrix, which means 100 cycles plus two for pipeline draining, totaling 102 cycles. It also needs to look at every element in the Timing Matrix, which has dimensions of 32 time steps maximum latency and four resource types, plus three cycles of pipeline draining, resulting in 131 cycles. This means the Resource Over-utilization Processor has a total latency of 232 cycles. As this is the most costly of the three sub-processors in the Evaluate Processor, the total latency of the Evaluate Processor is 233 cycles plus two for the final summations, resulting in a latency of 235 cycles.

The target device is a Xilinx Virtex-4 SX35 device, which consists of 15,360 slices, 192 DSP48 units, and 192 BRAM blocks. The design assumes 32-bit single-precision

floating-point arithmetic and 16-bit integer arithmetic. Single-precision floating point is needed to maintain the integrity of the temperature variable. Experiments with lower resolution and with fixed point representations resulted in an erratic and sometimes overly rapid temperature decline, which negatively influences the means by which simulated annealing avoids the pitfalls of local minima, as discussed in previous sections. A stage latency of the pipelined processor is 235 clock cycles, with a maximum clock frequency of 197 MHz (post place and route). At this speed, the entire iterative repair algorithm, consisting of 184,198 iterations can execute in just over 43 million clock cycles, or a wall-clock time of 220 ms. As shown in Table 2, this is a speedup of more than 850 times when compared to a PowerPC, without a floating-point coprocessor, running comparable code at 100 MHz.

Table 2: Comparative Results

Function	Number of Loads and Stores per function call		
	PowerPC	AMD Athlon	Custom Architecture
Copy	908	702	200
Alter	14	14	2
Evaluate	~15,000	~13,300	840
Accept	60	42	1
Adjust Temperature	10	5	2

While the PowerPC utilized was an embedded FPGA core, it uses a similar instruction set and the same basic pipeline architecture as the PowerPC 750 core generally used in space applications. The most significant difference between the two is the maximum clock frequency [21]. Furthermore, the custom architecture outperforms a desktop PC by a factor of 64.

Table 3: Load/Store Comparison

Processing Platform	Clock Freq.	Cycles	Time	Speedup
Xilinx Virtex-4 embedded PowerPC core	100 MHz	1.87×10^{10}	187.1 s	1.0
AMD Athlon 64	2.61 GHz	3.7×10^{10}	14.265 s	13.11
Xilinx Virtex-4 iterative repair circuit	197 MHz	4.32×10^{7}	220 ms	850.5

The reasons for the massive speed-up of the custom implementation when compared to traditional linear processors are three-fold. First, the custom circuit employs a four-stage macro pipeline. This allows for four different solutions to be at different stages of processing simultaneously, rather than only managing one solution at a time in the case of traditional processors. Second, the most complex of the processing stages, the evaluate function, has been parallelized in the custom implementation to drastically

decrease the latency of the pipeline. Once again, in a conventional processor, no such parallelization can occur. Third, in a conventional processor, up to 50 percent of the computation cycles in typical applications can be consumed by load and store instructions, especially in CISC architectures which have few internal registers. Because of the application-specific nature of the custom approach, no unneeded load/store cycles are consumed. Table 3 shows the load and store instructions used by each processor on a by-function basis. The custom architecture is by far more efficient in the utilization of load and store operations.

Based upon the results of Table 1 and the associated discussion, the performance of the custom architecture for larger problem sizes can be estimated. The size of the architecture will vary minimally for different sizes of input problems. A few additional address lines may be needed to address larger memories. Characterizing the performance in time is a much more interesting problem. In general, the ROP is the most costly with respect to time. The performance of the ROP can be characterized as shown in (2).

$$t_{ROP} = E + (L * R) + 7 \quad (2)$$

In (2), *E* represents number of events, *L* the maximum latency, *R* the number of resources, and t_{ROP} the number of clock cycles taken by the ROP processor. There is a total of seven cycles of delay associated with pipeline draining. Based upon this equation, Table 4 provides a performance estimate for the custom architecture compared to the PowerPC and the Athlon processor.

Table 3: Problem Size vs. Performance

Number of Events	Execution Time		
	PowerPC	AMD Athlon	Custom Architecture
100	187.1 s	14.625 s	220 ms
200	644.1 s	49.171 s	313 ms
400	2520.44 s	192.4 s	500 ms

Notice that as the problem size increases, the performance of the conventional processors begins drop off in an exponential manner. This is most-likely the result of a non-linear increase in the number of load and store operations introduced by the respective compilers. The custom processor, on the other hand, performs admirably for larger-sized problems. Because of the roughly linear characteristics of (2), handling larger event sets is not a problem for the custom architecture. In general, the larger the problem size, the more substantial the gain in execution time provided by the custom architecture.

There are a few differences between the software and custom hardware designs that need to be noted. First, in the pipelined custom hardware design, what should be done with solutions that are in the Alter and Evaluate stages when a new solution is accepted by the Accept stage? In the sequential software implementation, this issue does not exist, as

there is no high-level pipeline with multiple solutions in progress to worry about. This problem can be solved in the hardware implementation in one of two ways, either (1) flush the pipeline and start with a fresh solution, or (2) simply ignore the problem. In this architecture, we opted for solution 2 because of its simplicity. Even though the solutions in the Alter and Evaluate stages were created from a solution that is no longer the current solution, they are still valid potential solutions and can be treated as such. This saves the additional circuitry and delays needed to flush the pipeline. Because of this caveat, the custom hardware implementation may perform in a slightly different manner than the software version.

4. RESOURCE ESTIMATION AND RIPPLE LIST SCHEDULING

In addition to the architecture template discussed in the previous chapter, two additional concepts must be covered before the methodology for deriving custom architectures for iterative repair from C code can be presented. These two concepts are used as part of design space exploration for architecture generation. Resource estimation is used to predict how many FPGA resources (slices, DSP48s, BRAMs) would be used by an architecture. This estimation is much faster than generating VHDL code for every architecture and then running synthesis and place-and-route tools on each architecture to determine resource utilization. The second concept is a new form of scheduling for mapping control data flow graphs onto resources in an efficient manner.

4.1 Resource Estimation

Xilinx ISE is a tool for mapping high level hardware designs written using VHDL, Verilog, or schematic onto FPGAs and other hardware devices. The basic building block of an FPGA is the look-up table (LUT). In Xilinx FPGAs, two LUTs and associated logic form what is called a slice. Xilinx FPGAs also contain embedded 18 kb block RAM units (BRAMs) and embedded multiply-accumulate ASICs (DSP48s). One of the chief concerns of a hardware designer is ensuring that a hardware design will fit on a specific chip.

Running the Xilinx tools on a design is often time-consuming. Complex designs can take several hours to progress through the steps from synthesis to placing and routing. A mechanism for estimating resource utilization and permissible clock speeds without running these tools would greatly decrease design time.

One possible method for performing this estimation is to use curve fitting. Essentially, each type of design component is modeled for several different sizes of data inputs. Each model is sent through the Xilinx tool chain and final utilization values are determined. In this manner, a set of points are generated, where each point consists of an independent data parameter and dependent utilization parameter. A handful of these points scattered across the design space can be used as an input to a curve-matching algorithm (done using Matlab), in which a high-order polynomial equation can be derived which approximates the curve represented by the design points.

For example, consider an integer addition unit. For simplicity, it is assumed that the adder takes in two n-bit numbers and produces a single n-bit output, registering the output and consuming a single clock cycle. The goal is to derive a function that relates data width (n) to slice consumption and maximum clock speed. To determine this, several different sizes of adders are instantiated using Xilinx ISE, noting the post-place-and-route utilization statistics for each one, including slices used and maximum allowable clock frequency. This data is shown in table 5. When using ISE to obtain timing statistics, it is important to measure the delay of the circuit only. By default, ISE maps HDL I/O ports to physical I/O buffers on the FPGA, resulting in significant delays that

are not actually part of the circuit under test itself. This problem can be avoided by unselecting the "Add IO Buffers" option in the ISE synthesis menu.

Table 5: Measured resource utilization and clock speed for discrete sizes of integer adders.

Data Width	Slices	Maximum Clock Freq. (MHz)
1	1	1381
2	1	1381
4	3	712
8	4	676
12	6	619
16	8	571
24	12	494
32	16	436

Using this data, best-fit equations can be derived for both slice utilization and maximum allowable clock frequency. A fifth-order polynomial, generated by Matlab, is deemed sufficient for curve approximation. This fifth-order polynomial is shown in (3).

$$y = C_5n^5 + C_4n^4 + C_3n^3 + C_2n^2 + C_1n + C_0 \quad (3)$$

The goal is to find the value of all coefficients (C_x). Using the Matlab curve-fitting function, the best-fit 5th-order polynomial for slice usage in an integer adder is shown in (4).

$$y = 4.22e\text{-}6n^5 - 3.46e\text{-}4n^4 + 1.01e\text{-}2n^3 - 1.24e\text{-}1n^2 + 1.05e{+}0n - 1.74e\text{-}1 \quad (4)$$

This equation can then be used to predict slice utilization for an integer adder of arbitrary size. The correlation between the measured and computed values is shown in fig. 19. It is an almost-perfect match.

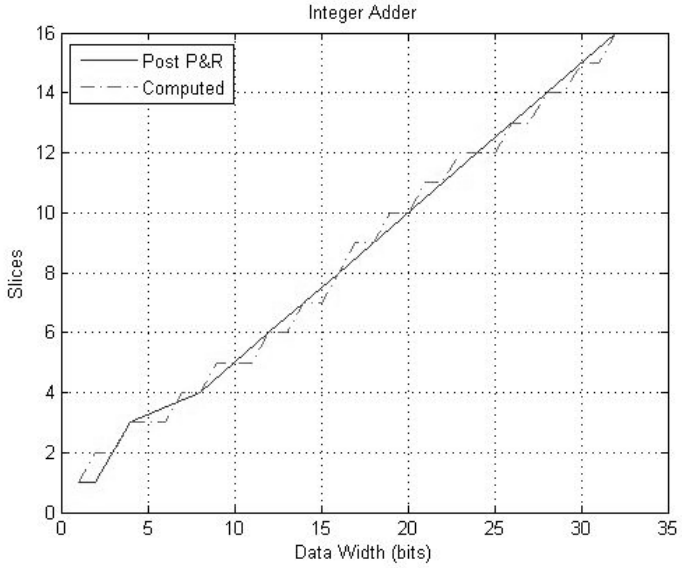


Figure 19: Slice consumption for integer adders.

A similar equation and plot can be generated for maximum clock frequency. These are shown in (5) and fig. 20, respectively.

$$y = -1.22e\text{-}3n^5 + 1.10e\text{-}1n^4 - 3.73e+0n^3 + 5.86e+1n^2 - 4.34e+2n + 1.85e+3 \quad (5)$$

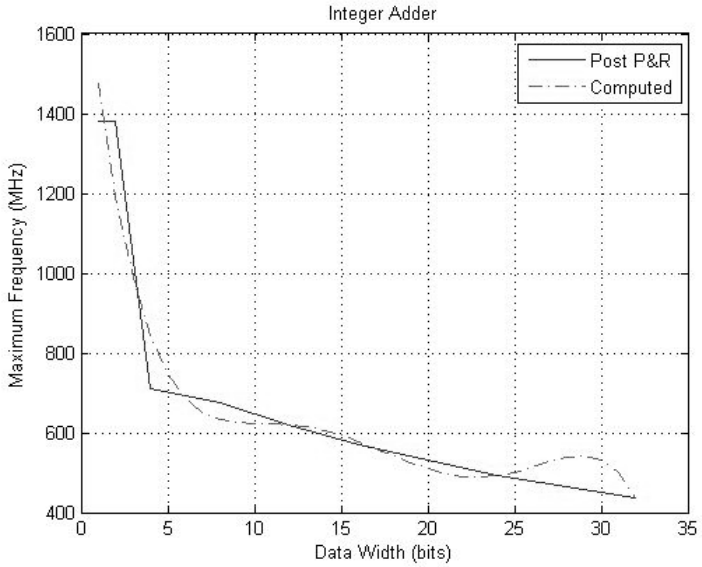


Figure 20: Maximum clock frequency for integer adders.

This technique can be repeated for any module in which resource utilization is a function of data width. Table 6 shows the data that has been derived for integer multipliers of varying data widths, once again with a single-cycle clock latency.

Table 6: Measured resource utilization and clock speed for discrete sizes of integer multipliers.

Data Width	Slices	DSP48s	Maximum Clock Freq. (MHz)
1	1	0	1381
2	1	0	1381
4	4	0	461
8	0	1	277
12	0	1	277
16	0	1	277
24	0	3	128
32	0	3	128

In addition to slices, integer multipliers use DSP48 resources on the FPGA. Equations are derived for slice utilization (6), DSP48 utilization (7), and maximum clock frequency (8).

$$y = 3.28e\text{-}5n^5 - 2.68e\text{-}3n^4 + 7.80e\text{-}2n^3 - 9.61e\text{-}1n^2 + 4.47e+0n - 3.20e+0 \quad (6)$$
$$y = 4.22e\text{-}6n^5 - 3.46e\text{-}4n^4 + 1.01e\text{-}2n^3 - 1.24e\text{-}1n^2 + 1.05e+0n - 1.74e\text{-}1 \quad (7)$$
$$y = -4.73e\text{-}4n^5 + 6.25e\text{-}2n^4 - 2.90e+0n^3 + 5.93e+1n^2 - 5.36e+2n + 1.99e+3 \quad (8)$$

Fig. 21, 22, and 23 show the quality of the estimation for slices, DSP48 units, and maximum clock frequency, respectively. Estimations are generally very accurate.

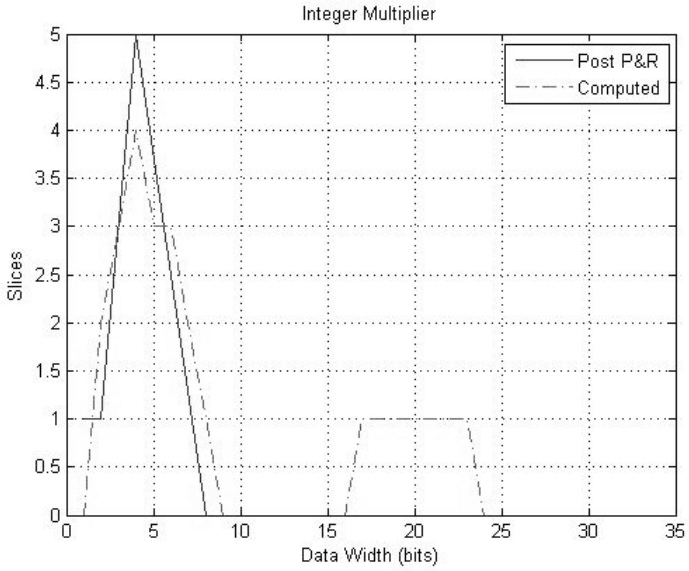


Figure 21: Slice utilization for integer multiplier.

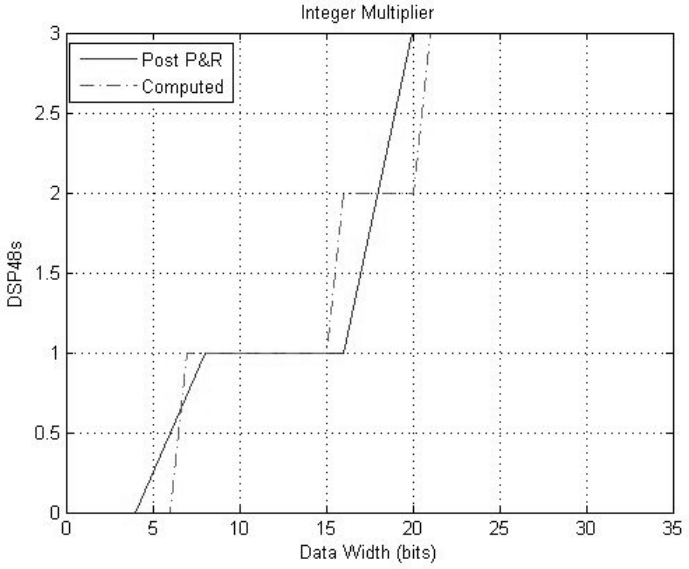


Figure 22: DSP48 utilization for integer multiplier.

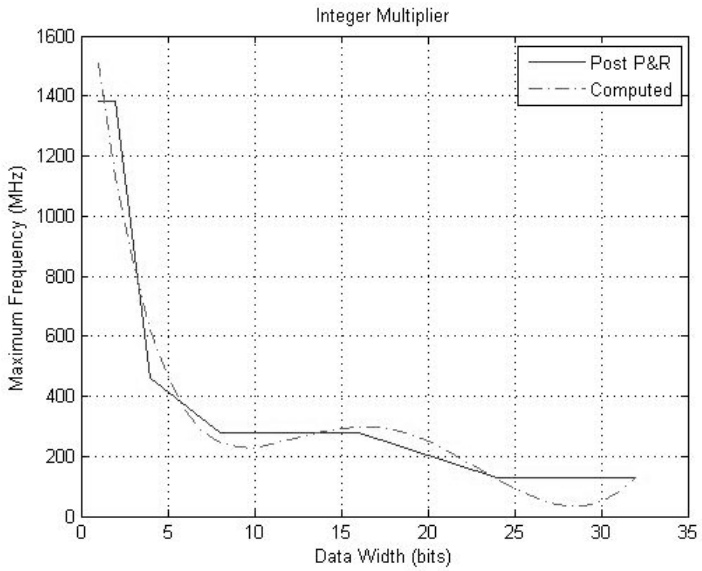


Figure 23: Maximum clock frequency for integer multiplier.

A set of equations for multiplexers is also derived. The number of inputs to a traditional multiplexer is generally a power of 2 (2, 4, 8, 16, 32, etc.). In many cases, not all input lines may be used. In theory, creating a multiplexer with 31 inputs should use the same resources as one with 32. Due to suboptimal analysis by the Xilinx tools, however, this was found to be an incorrect assumption in general. The measured slice utilization data for multiplexers is given in table 7.

Table 7: Measured resource utilization and clock speed for discrete sizes of multiplexers.

Number of inputs	Slices	Maximum Clock Freq. (MHz)
2	1	2299
3	1	1377
4	1	1377
5	2	912
6	2	912
7	2	975
8	2	975
9	3	625
10	4	625
11	4	721
12	5	598
13	4	598
14	4	598
15	4	754
16	4	754
17	6	458
18	8	442
19	8	404
20	8	436
21	7	442

22	9	415
23	9	375
24	10	387
25	7	507
26	8	507
27	9	441
28	9	441
29	9	441
30	9	450
31	8	615
32	8	615

The equations for modeling the slice usage and maximum clock frequency of multiplexers are given in (9) and (10), respectively.

$$y = 4.18e\text{-}6n^5 - 3.72e\text{-}4n^4 + 1.12e\text{-}2n^3 - 1.36e\text{-}1n^2 + 9.63e\text{-}1n - 8.12e\text{-}1 \quad (9)$$

$$y = -1.20e\text{-}3n^5 + 1.21e\text{-}1n^4 - 4.52e+0n^3 + 7.98e+1n^2 - 6.89e+2n + 3.05e+3 \quad (10)$$

The performance of the estimation for the slice utilization is shown in fig. 24. Even for this odd-shaped graph with several discontinuities, a fifth-order polynomial provides an acceptable estimation. The frequency performance graph is also shown in fig. 25.

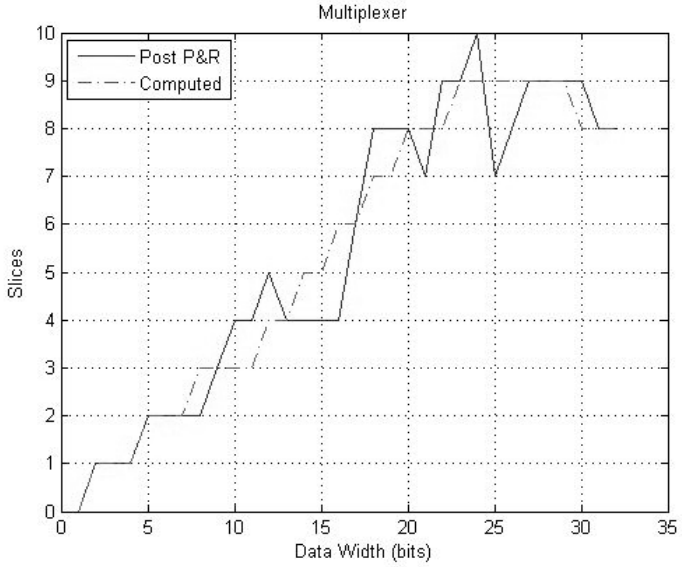


Figure 24: Slice utilization for multiplexers.

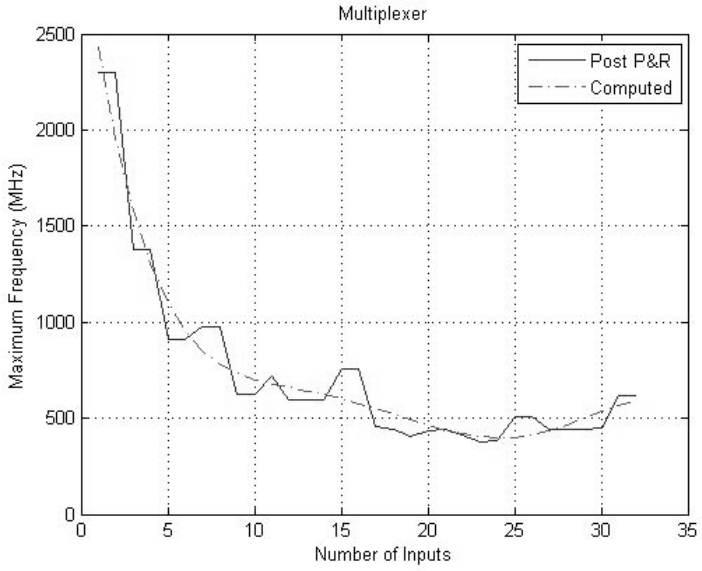


Figure 25: Maximum clock frequency for multiplexers.

After characterization of several iterative repair and other simulated annealing algorithms, a set of nine basic modules has been identified for parameterization. These nine modules are multiplexers, registers, random number generators (RNG), and six integer arithmetic operators (adder, subtractor, multiplier, divider, modulo arithmetic, and absolute value (ABS)). All modules are assigned a latency of one clock cycle, with the exception of the divider and modulo units, which take 19 cycles, and the multiplexers, which are not clocked. Table 8 provides the fifth-order coefficients for resource utilization equations and clock frequency equations for all nine modules. Note that all modules use slices, while only multipliers employ the use of DSP48 units.

Table 8: Fifth-order equation coefficients for all supported blocks.

Parameter	C_5	C_4	C_3	C_2	C_1	C_0
adder slices	4.2210 e-006	-3.4624 e-004	1.0094 e-002	-1.2379 e-001	1.0502 e+000	-1.7433 e-001
adder frequency	-1.2180 e-003	1.1023 e-001	-3.7314 e+000	5.8622 e+001	-4.3409 e+002	1.8534 e+003
subtractor slices	4.2210 e-006	-3.4624 e-004	1.0094 e-002	-1.2379 e-001	1.0502 e+000	-1.7433 e-001
subtractor frequency	-1.2180 e-003	1.1023 e-001	-3.7314 e+000	5.8622 e+001	-4.3409 e+002	1.8534 e+003
multiplier slices	3.2830 e-005	-2.6831 e-003	7.8032 e-002	-9.6178 e-001	4.4677 e+000	-3.2061 e+000
multiplier DSP48s	2.2727 e-006	-2.0191 e-004	6.1369 e-003	-7.4108 e-002	4.2923 e-001	-5.2696 e-001
multiplier frequency	-4.7294 e-004	6.2525 e-002	-2.9023 e+000	5.9290 e+001	-5.3634 e+002	1.9880 e+003
divider slices	0	0	0	0	0	3.2200 e+002

divider frequency	0	0	0	0	0	3.3000 e+002
modulo slices	0	0	0	0	0	3.2200 e+002
modulo frequency	0	0	0	0	0	3.3000 e+002
RNG slices	0	0	0	0	0	9.0000 e+000
RNG frequency	0	0	0	0	0	1.0380 e+003
ABS slices	7.5318 e-008	1.0667e -004	-7.7516 e-003	1.7508 e-001	-2.4127 e-001	6.8765 e-002
ABS frequency	-1.1498 e-002	9.5933 e-001	-2.9028 e+001	3.8969 e+002	-2.2666 e+003	4.9340 e+003
register slices	0	0	0	0	0	1.0000 e+000
register frequency	0	0	0	0	0	3.4600 e+003
multiplexer slices	4.1773 e-006	-3.7247 e-004	1.1232 e-002	-1.3629 e-001	9.6286 e-001	-8.1262 e-001
multiplexer frequency	-1.2063 e-003	1.2064 e-001	-4.5216 e+000	7.9830 e+001	-6.8925 e+002	3.0466 e+003

Now that the basic building blocks have been parameterized, the next question is to determine how well this method will estimate circuits comprised of two or more of these blocks. In theory, resource consumption should be a roughly additive property, while maximum clock frequency should be close to that of the worst-performing block. As a simple example the circuit shown in fig. 26 is considered.

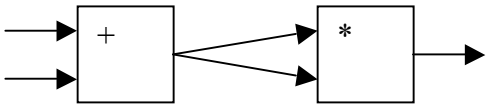


Figure 26: Simple circuit with an addition operation followed by a squaring operation.

Using the blocks described above, the circuit of fig. 26 takes two clock cycles to produce a result. This circuit can be implemented with various data bit widths. Table 9 shows the results for bit widths ranging from 4 to 24.

Table 9: Computed vs. actual values for simple add/multiply circuit.

Data width	Slice Utilization		DSP48 Utilization		Maximum Clock Freq. (MHz)	
	Computed	Actual	Computed	Actual	Computed	Actual
4	8	5	0	0	461	552
8	4	4	1	1	277	417
12	6	6	1	1	277	395
16	8	8	1	1	277	375
24	12	12	3	3	128	133

Notice that the slice utilization and DSP48 utilization predictions are reasonably close to the actual post-place-and-route values for all data widths. The predicted maximum clock frequencies, on the other hand, are substantially lower than the actual values, especially in circuits with smaller data widths. It is obvious that Xilinx ISE must do some additional optimization of the circuit as a whole to reduce the delay. This additional optimization is very hard to characterize.

For a more-complex example, let consider the circuit depicted in fig. 27. The resource utilization and timing statistics are shown in table 10.

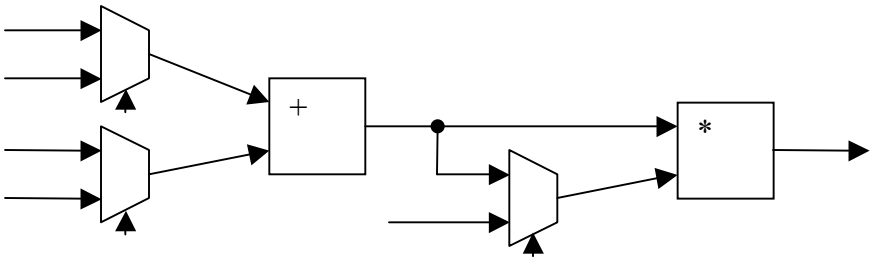


Figure 27: Multi-functional add/multiply circuit.

Table 10: Resource utilization and timing statistics for complex add/multiply circuit.

Data width	Slice Utilization		DSP48 Utilization		Maximum Clock Freq. (MHz)	
	Computed	Actual	Computed	Actual	Computed	Actual
4	20	13	0	0	461	338
8	28	13	1	1	277	244
12	42	19	1	1	277	244
16	56	25	1	1	277	244
24	84	37	3	3	128	115

The actual slice utilization in this example is consistently lower than the computed values. This suggests that optimizations are mode to share resources between sub-modules. Also, delay estimates are not even close to the actual numbers, although the same trend of closer matching with increasing data widths applies to this circuit as well.

For an extreme case of one adder feeding another, the circuit depicted in fig. 28 is considered. This pipelined circuit has a latency of six cycles. Numerical results are shown in table 11.

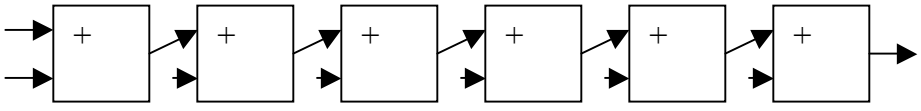


Figure 28: A 6-adder chain circuit.

Table 11: Resource utilization and timing statistics for the 6-adder chain.

Data width	Slice Utilization		DSP48 Utilization		Maximum Clock Freq. (MHz)	
	Computed	Actual	Computed	Actual	Computed	Actual
4	18	21	0	0	712	635
8	24	24	0	0	676	573
12	36	36	0	0	619	531
16	48	48	0	0	571	495
24	72	72	0	0	494	436

In this circuit, the computed and actual resource utilization is almost identical for all cases. The actual maximum clock frequency is consistently higher than predicted.

In conclusion, this method of approximating FPGA performance seems to perform reasonably well for estimating resource utilization. It can be concluded that resource usage is generally an additive property for more-complex circuits built from simple parameterized blocks. Timing performance, on the other hand, is not so easy to characterize. In any pipelined or clocked circuit, the maximum clock frequency is constrained by the worst-case critical path between any two registers in the circuit. In theory, the lowest maximum allowable frequency of all of the simple blocks in a circuit would be the maximum clock frequency of that circuit. However, because of optimizations performed by the synthesis tools, complex circuits consistently perform better than the individual parts. A complete set of plots of resource usage and frequency performance for all nine of the targeted modules discussed in this section is provided in Appendix A.

4.2 Ripple List Scheduling

4.2.1 Overview

To estimate the temporal performance of an architecture, a mapping must take place of the operations to be performed onto the available resources. For example, fig. 29 depicts a set of 7 arithmetic operations (addition, multiplication, and division) and their associated dependencies. This dataflow graph can be scheduled on an architecture that consists of assorted addition, multiplication, and division units.

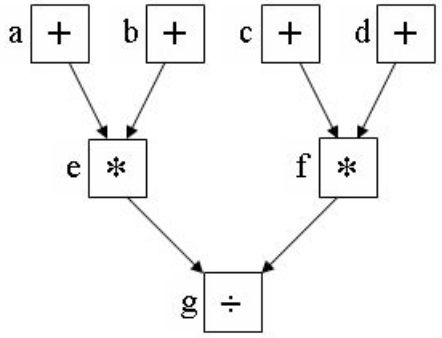


Figure 29: A simple dataflow graph.

A standard method for performing these types of scheduling problems is List Scheduling [70]. The basic List Scheduling algorithm, also known as Critical Path Scheduling, consists of assigning a static priority to each node in the graph and scheduling the nodes according to priority. These static priorities are assigned by measuring the "distance" from the node in question and a sink node. For example, in fig. 29, if multiplication and division took two time units and addition one time unit, node "g" would be assigned a priority of 2, "e" and "f" a priority of 4, and "a", "b", "c", and "d" a priority of 5. Once priorities have been assigned, nodes are successively scheduled by availability and highest priority. In the example of fig. 29, if a resource set of one adder, one multiplier, and one divider is assumed, any one of "a", "b", "c", or "d" could be selected to run in the first time slot. Once "a" and "b" have completed, "e" could be scheduled. One possible list scheduling for fig. 29 is shown in table 12.

Table 12: A List Schedule for the DFG shown in fig. 19.

Time Step	Adder	Multiplier	Divider
0	a		
1	b		
2	c	e1	
3	d	e2	
4		f1	
5		f2	
6			g1
7			g2

In this case, the entire dataflow graph can be scheduled in eight time steps. This happens to be an optimal schedule for this operation/resource pairing. No matter how the operations are scheduled, a valid schedule that takes less than eight steps cannot be found.

Unfortunately, Critical Path Scheduling is not guaranteed to find an optimal schedule. Considering the example of fig. 29 once again, it is equally possible that the list scheduler could come up with the schedule shown below in Table 13.

Table 13: An alternative List Schedule for the DFG shown in fig. 19.

Time Step	Adder	Multiplier	Divider
0	a		
1	c		
2	b		
3	d	e1	
4		e2	
5		f1	
6		f2	
7			g1
8			g2

Because “c” was selected to occupy the adder during time step 1, “e” was delayed until step 3, which delayed “f” to step 5 and “g” to step 7. An extra time step is needed to complete all of the computations. More broadly, there are 4!, or 24, different orders in which a list scheduler can schedule “a”, “b”, “c”, and “d”. Of those 24 permutations, only 8 will produce the optimal schedule. In other words, in this simple example there is only a 1 in 3 chance that List Scheduling will perform properly. In more complex scheduling problems with more nodes, more edges, more resources, and pipelining constraints, more opportunities for choosing suboptimal operations will occur and the odds of stumbling across the optimal schedule will decrease even more.

While List Scheduling has been shown to be a suboptimal algorithm, it has one nice characteristic that can be leveraged to design a better scheduler – List Scheduling is fast. In fact, it has a computational complexity of only $O(Tn)$, where T is the number of time slots and n is the number of nodes to be scheduled. Other scheduling algorithms, such as Force-Directed Scheduling or Force-Directed List Scheduling, derive more-efficient schedules, but involve the repeated multi-step re-computation of priority values for each remaining node every time a node is scheduled, resulting in more-complex algorithms. For example, Force-Directed List Scheduling is an $O(n^2)$ algorithm. List Scheduling, on the other hand, requires only an initial static priority assignment.

Many improvements and modifications to the basic List Scheduling algorithm, such as Modified Critical Path [71, 72], Earliest Time First [73], Dynamic Critical Path [74], topological clustering [75], Critical Node Parent Trees [76], Cone-Based Clustering[77], and Partial Critical Path scheduling [78], have been proposed over the years. These algorithms improve the performance of the basic List Scheduling algorithm at the expense of increasing algorithm complexity. Different techniques are employed for each type of scheduling.

For example, Modified Critical Path scheduling requires re-computing the critical path after every node scheduling event. Every time a node is scheduled, the critical path through the remaining nodes may change. This method improves schedule optimality. The time complexity of this algorithm is $O(n^2 \log n)$. A “brute-force” version of Modified Critical Path scheduling is Dynamic Critical Path scheduling, with a complexity of $O(n^3)$.

The Earliest Time First algorithm requires that earliest start times for all ready nodes be computed on each scheduling step. The earliest start times are computed by finding the distance from the sink node. The node with the lowest start time is selected. This algorithm requires $O(n^2)$ work to schedule each of n nodes, making this also an $O(n^3)$ algorithm.

All surveyed modifications to the List Scheduling algorithm have time complexities ranging from $O(n^2)$ to $O(n^3)$, which are too time-consuming for use inside of a design-space explorer where the scheduler could be run hundreds or thousands of times on very large graphs. A more-efficient algorithm is needed that runs in better than $O(n^2)$ time.

4.2.2 Proposed Algorithm

The proposed algorithm provides a flexible manner of improving the basic Critical Path List Scheduling algorithm. The tradeoff between algorithmic complexity and solution quality can be adjusted through the modification of one simple parameter. For reasons that will be described below, this new method will be termed *Ripple List Scheduling*. The Ripple List Scheduling algorithm consists of (1) identifying sink nodes and assigning static node priorities in the same manner as done by Critical Path List Scheduling, (2) repeatedly scheduling the node with the highest priority in the ready list on the current time step, and (3) updating the priority of remaining nodes in a dynamic manner based upon distance from the scheduled node. Pseudocode that describes this algorithm in detail is shown below:

```
assign static priority to each node in graph
initialize time to 0
Loop while unscheduled nodes exist
        Loop until no nodes can be scheduled on time step
                update list of ready nodes
                schedule highest priority node possible
                adjust priority of remaining nodes
        EndLoop
        increment time
EndLoop
```

The only major difference between this algorithm and classic List Scheduling is the priority adjustment on remaining nodes after every node has been scheduled. Three things must be known to adjust the priority of a node: first, the distance between the node in question and the recently scheduled node; second, the *ripple factor*; and third, the *maximum ripple distance*.

In order to compute the ripple factor, let us first borrow the definition of the *degree* of a vertex from graph theory. The degree of a vertex is the number of edges (both incoming and outgoing in the case of a directed graph) incident to it. The largest vertex degree in the entire graph will be designated the graph degree, represented as D_G. Also, let us define the distance between two vertices as the minimum number of edge traversals needed to move from one to the other (edge directionality is ignored in this case), represented as d.

The ripple factor, defined as R_f, is computed as shown in equation 11.

$$R_f = \frac{1}{D_G^d} \tag{11}$$

The ripple factor for a given node is added to the priority of that node to create a new priority level. This priority level can change every time any node in the graph is scheduled. As can be seen from the equation, the ripple factor is inversely proportional

to the distance from the scheduled node. This dynamic priority adjustment can be thought of as a ripples propagating out from a stone dropped in a pond. The ripple size gradually decreases as it travels farther from the source. The ripple factor is computed as such to allow for variations in dynamic priority between nodes with the same static priority, while at the same time preventing the dynamic priority from encroaching upon nodes with a higher static priority. In other words, nodes with a static priority of three will end up with a dynamic priority greater than or equal to three, but guaranteed to be less than four, thus avoiding conflicts with nodes with static priority of four. Fig. 30 shows an example of this ripple effect on the graph introduced in fig. 29.

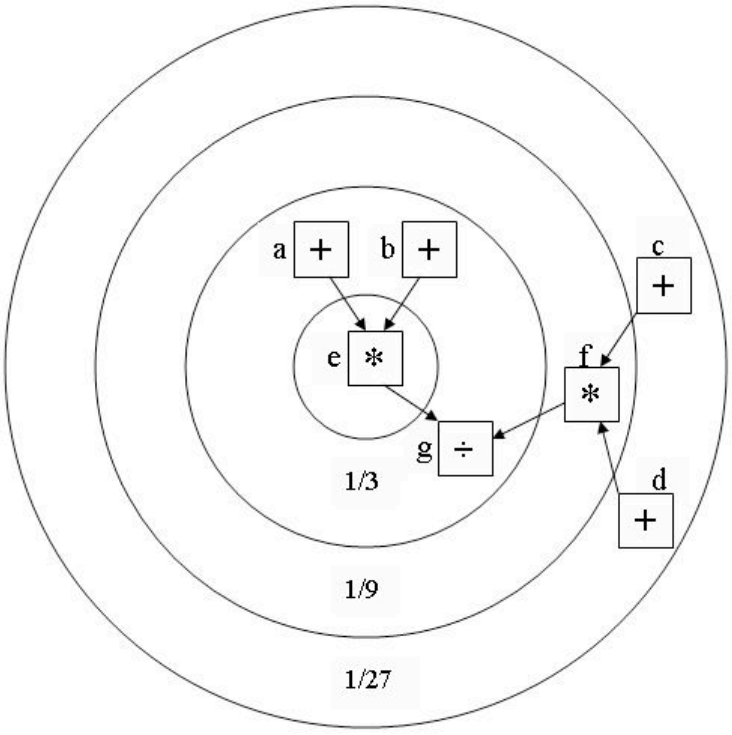


Figure 30: An example of ripple factor propagation in the Ripple List Scheduling algorithm.

In fig. 30, it is assumed that node "e" is being scheduled. Disregarding the feasibility of "e" being scheduled before "a" and "b", all other nodes in the graph would be updated by the specified amounts. The graph degree in this case is 3, as no vertex has more than 3 edges incident to it. Nodes that are one step away get updated by a ripple factor of $1/3^1$, those that are two steps away get updated by $1/3^2$, and so forth.

As stated before, the maximum ripple distance can be used to balance the performance of the algorithm between execution time and solution quality.

4.2.3 Performance

The Ripple List Scheduling algorithm can be compared with standard Critical Path List Scheduling. For all examples in this section, addition and subtraction operations take one clock cycle; multiplication and division take two. Both multiplication and division units

are pipelined (i.e. new data can be accepted on each clock cycle). The Critical Path List Scheduling algorithm was performed 1000 times on each DFG and the mean schedule length and standard deviation are computed in each case. Fig. 31 shows the correlation between colors in the DFG representations and arithmetic operators.

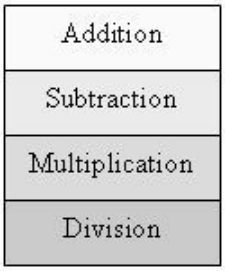


Figure 31: Color coordination for operations shown in data flow graphs.

Let us first consider the simple DFG shown in fig. 32.

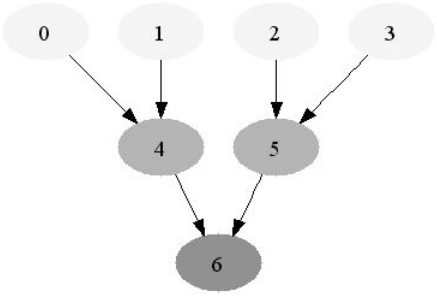


Figure 32: A simple 8-node DFG.

This DFG has been scheduled for a variety of different resource sets. Results are shown in Table 14. Columns are present for Critical Path List Scheduling (CPLS) and three versions of Ripple List Scheduling (RLS). The number following the RLS designation in parentheses is the maximum ripple distance. In this simple example, all four scheduling techniques derive an optimal schedule.

Table 14: Schedules for the DFG shown in fig. 22.

+	*	÷	CPLS Schedule Length	RLS(1) Schedule Length	RLS(2) Schedule Length	RLS(3) Schedule Length
1	1	1	8	8	8	8
1	2	1	8	8	8	8
2	2	1	6	6	6	6

Next, let us consider the larger graph shown in fig. 33. The scheduling results for this graph are shown in Table 15.

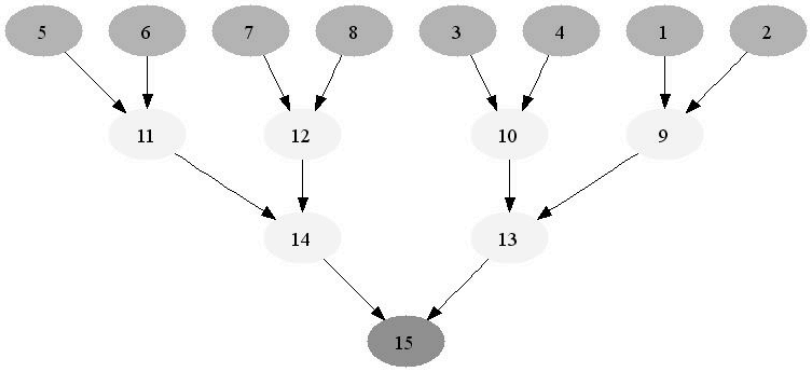


Figure 33: A 15-node, 3-operation-type DFG.

Table 45: Schedules for the DFG shown in fig. 23.

+	*	÷	CPLS Schedule Length	RLS(1) Schedule Length	RLS(2) Schedule Length	RLS(3) Schedule Length
1	1	1	x = 14.0, σ = 0.0	14	14	14
1	2	1	x = 11.25, σ = 0.43	10	10	10
1	3	1	x = 10.56, σ = 0.49	10	10	10

In Table 15, the mean CPLS schedule is given along with a standard deviation. Notice that as the number of resources is increased, RLS begins to perform better than CPLS. The other nice characteristic is that RLS generates the same schedule every time it is executed on the same problem. There is no need to worry about means and standard deviations. Even when the maximum ripple distance is restricted to one, an immediate improvement is gained in two of the three cases shown.

The next example is shown in fig. 34. This DFG appears to be a good candidate for extracting parallelism. The scheduling results are shown in Table 16.

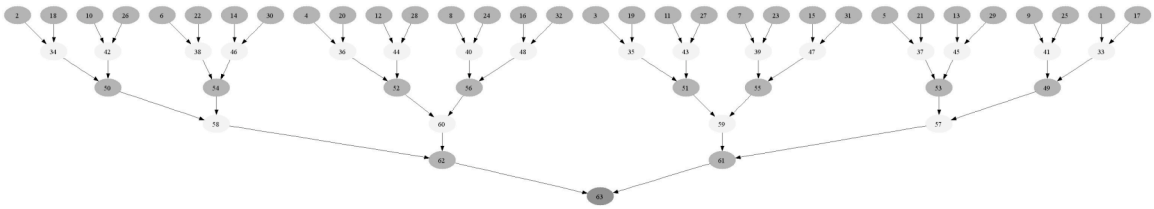


Figure 34: A 63-node, 3-operation-type DFG.

Table 16: Scheduling results for the DFG shown in fig. 24.

+	*	÷	CPLS Schedule Length	RLS(1) Sched. Length	RLS(2) Sched. Length	RLS(3) Sched. Length	RLS(4) Sched. Length	RLS(5) Sched. Length
1	1	1	x = 46.0, σ = 0.0	46	46	46	46	46
1	2	1	x = 30.60, σ = 0.86	34	26	28	26	26
1	3	1	x = 28.26, σ = 0.67	31	26	26	26	26
2	3	1	x = 20.96, σ = 0.40	22	20	20	20	20
3	5	1	x = 16.8, σ = 0.39	18	16	16	16	16

Once again, the RLS algorithm performs better than CPLS in general. Notice that RLS(1), however, actually performs worse. This suggests that RLS(1) should generally be avoided. In one case, RLS(3) also yields sub-optimal performance. RLS(4) appears to be a safe bet.

The final example, shown in fig. 35, is a deeper graph than the one shown in fig. 34. The scheduling results are shown in Table 17.

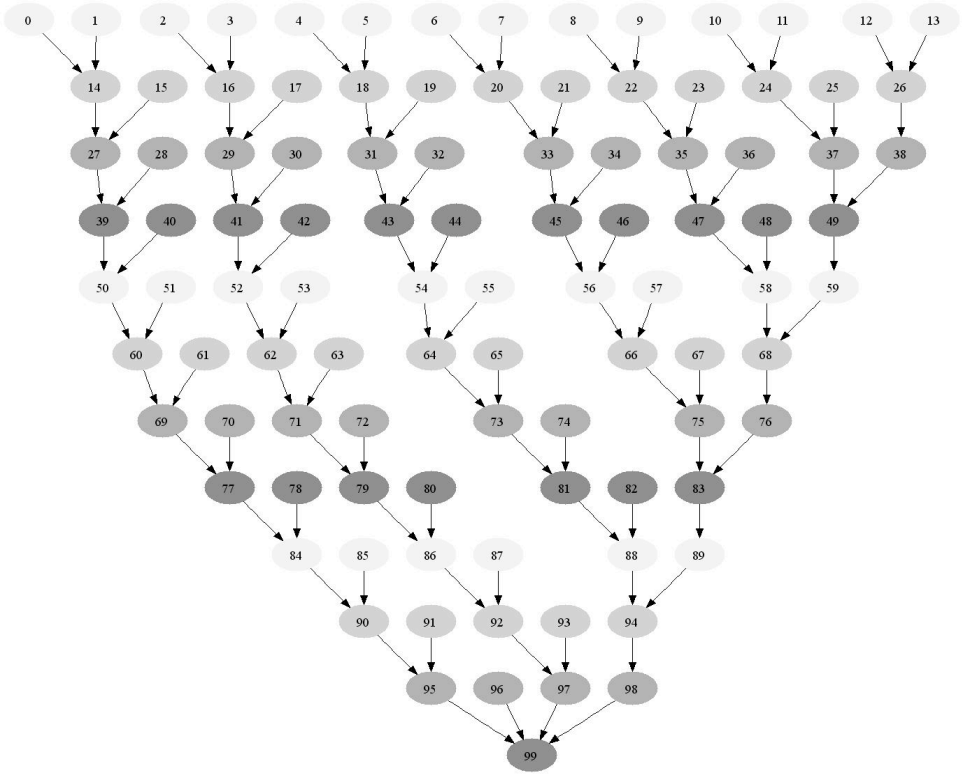


Figure 35: A 100-node, 4-operation-type DFG.

Table 17: Scheduling results for the DFG shown in fig. 25.

+	-	*	÷	CPLS Schedule Length	RLS(1) Sched. Length	RLS(2) Sched. Length	RLS(3) Sched. Length	RLS(4) Sched. Length	RLS(5) Sched. Length
1	1	1	1	$x = 35.0, \sigma = 0.0$	35	35	35	35	35
1	1	1	1	$x = 32.0, \sigma = 0.0$	32	32	32	32	32
7	7	6	6	$x = 19.04, \sigma = 0.21$	19	19	19	19	19
7	7	3	3	$x = 20.04, \sigma = 0.21$	20	20	20	20	20

Once again, RLS performs as well as or better than CPLS in all cases. In this example, all versions of RLS perform in an identical manner.

In summary, Ripple List Scheduling performs as well as or better than traditional Critical Path List Scheduling in all cases. At the same time, it is much less compute-intensive than other algorithms proposed over the years. Rather than the $O(n^2)$ to $O(n^3)$ complexity

required by these algorithms, RLS has a variable complexity which is a function of the maximum ripple distance. Adequate results can be achieved with a maximum ripple distance of only two or three steps, making RLS an algorithm of complexity O(*nm*), where *m* is the maximum ripple distance.

5. ARCHITECTURE DERIVATION

In this section, a novel algorithm and associated tool flow (SATH) are described for generating FPGA-based application specific processors for simulated annealing algorithms. Fig. 36 shows a flow chart of the different steps in the tool. First, C code to be converted is provided to the system, along with architecture templates for different components. Next, the intermediate representation (IR) is interpreted and divided into modules. In each module, the IR is mapped to the associated template. Finally, the derived architectures are combined and translated into synthesizable VHDL code. Details of the simulated annealing template and the tool flow are discussed here.

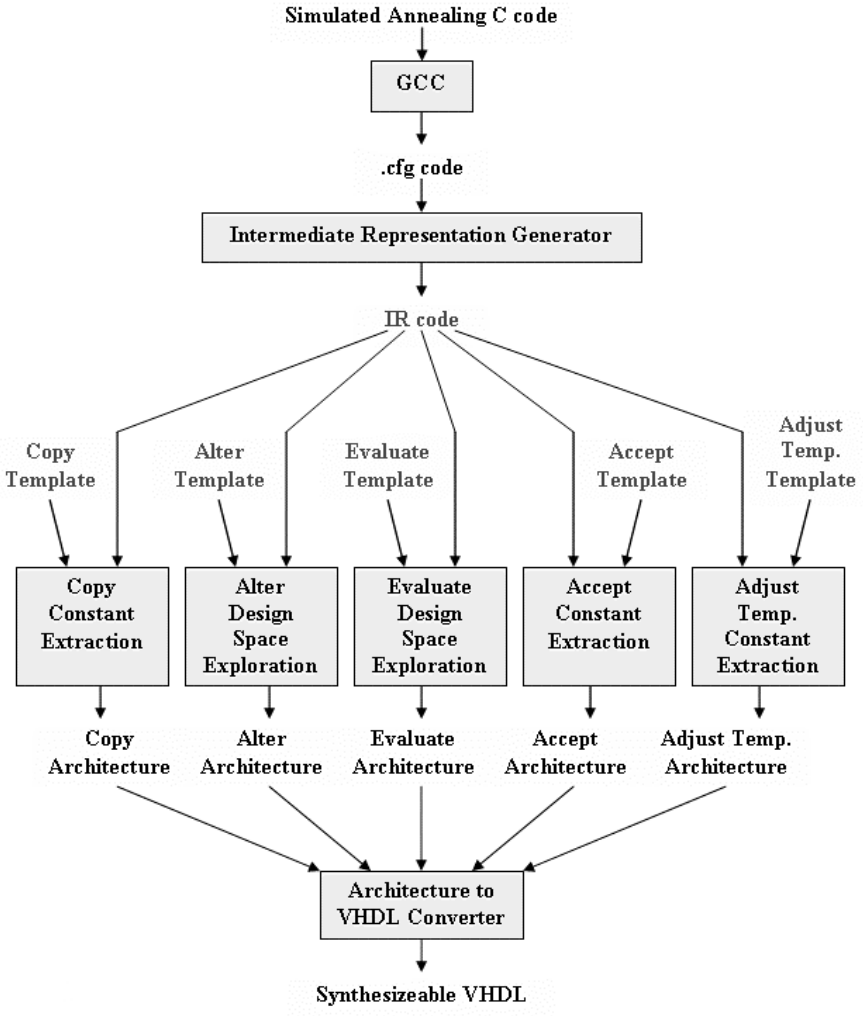


Figure 36: The SATH Tool flow.

In order to facilitate the discussion of the SATH tool flow, the classic Traveling Salesperson Problem (TSP) is used. Briefly, TSP is defined as a minimization problem in which a salesperson must visit n cities in any order, the goal being to minimize the total distance the salesperson must travel. In this example, 100 cities are considered; each assigned a random (x, y) location on a 100 by 100 grid. Distances between cities are measured using Manhattan distance (distances are measured on a grid, rather than

straight-line). A simple geometric cooling schedule is assumed. A solution is represented by keeping an array of the order in which cities are visited. The array indices represent the order of visitation, while the entries represent the cities. Separate static arrays store the location of each city on the grid.

5.1 Simulated Annealing Template

As discussed in Chapter 3, the simulated annealing algorithm lends itself well to implementation as an application-specific, coarse-grained pipelined processor. The block diagram of the processor template derived in Chapter 3 is duplicated here for convenience, shown in fig. 37.

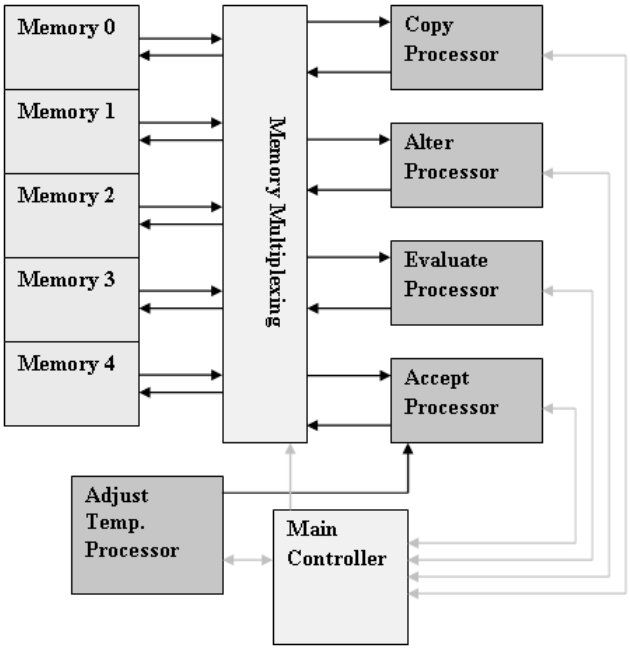


Figure 37: The simulated annealing design template.

The template is composed of a four-stage pipeline coupled with five memory banks. Each stage in the pipeline corresponds to a step in the simulated annealing pseudocode (fig. 1) – copy, alter, evaluate, and accept. A global controller coordinates execution and data exchange between the units. An interface between memory banks and processors is provided. An Adjust Temperature Processor controls the cooling process. As this is a pipelined architecture, it can only operate as fast as the slowest stage. At any given point in execution time, one memory bank is associated with each of the four processing stages in the pipeline. The remaining memory block holds the current solution. The main controller determines how memory blocks are associated with different processing stages.

5.2 C Code to Intermediate Format

The first stage in the SATH tool uses the front end of the GNU gcc compiler. The gcc compiler provides extensive options for dumping various intermediate formats to files. For SATH, the ".cfg" format is generated. This format is a single-assignment three-

address code, similar in structure to C. This code is then passed through the Intermediate Representation Generator portion of SATH to produce a custom intermediate representation that is a low-level control data flow graph (CDFG). Fig. 38 shows the graphical output of this stage for the Alter portion of the TSP example. The C code for this portion is shown here.

```
void alter(int *next) {
   int a, b, temp;

   a = rand() % MAX_EVENTS;
   b = rand() % MAX_EVENTS;
   temp = next[a];
   next[a] = next[b];
   next[b] = temp;
}
```

The Intermediate Representation Generator (IRG) is actually a two-pass compiler stage. In the first stage, the ".cfg" format taken from gcc is parsed. For the Alter function C code listed above, the generated ".cfg" file is as follows.

```
alter (next)
{
 int temp;
 int b;
 int a;
 int * D.3031;
 int * D.3030;
 unsigned int D.3029;
 unsigned int b.8;
 int D.3027;
 int * D.3026;
 int * D.3025;
 unsigned int D.3024;
 unsigned int b.7;
 int * D.3022;
 int * D.3021;
 unsigned int D.3020;
 unsigned int a.6;
 int * D.3018;
 int * D.3017;
 unsigned int D.3016;
 unsigned int a.5;
 int D.3014;
 int D.3013;

 # BLOCK 0
 # PRED: ENTRY (fallthru)
 D.3013 = rand ();
```

```
  a = D.3013 % 100;
  D.3014 = rand ();
  b = D.3014 % 100;
  a.5 = (unsigned int) a;
  D.3016 = a.5 * 4;
  D.3017 = (int *) D.3016;
  D.3018 = D.3017 + next;
  temp = *D.3018;
  a.6 = (unsigned int) a;
  D.3020 = a.6 * 4;
  D.3021 = (int *) D.3020;
  D.3022 = D.3021 + next;
  b.7 = (unsigned int) b;
  D.3024 = b.7 * 4;
  D.3025 = (int *) D.3024;
  D.3026 = D.3025 + next;
  D.3027 = *D.3026;
  *D.3022 = D.3027;
  b.8 = (unsigned int) b;
  D.3029 = b.8 * 4;
  D.3030 = (int *) D.3029;
  D.3031 = D.3030 + next;
  *D.3031 = temp;
  return;
  # SUCC: EXIT
}
```

Notice both the similarities and differences between this intermediate format and general C. The single-assignment characteristics elongate the code significantly. This code is then parsed by IRG and a text-based control data flow graph is created. The control data flow graph representation of the code above is shown below.FUNCTION alter
BASIC_BLOCK 0
DATA_NODE 0 FUNCTION rand
DATA_NODE 1 MOD integer
CONST_NODE 0 100
DATA_NODE 2 FUNCTION rand
DATA_NODE 3 MOD integer
CONST_NODE 1 100
DATA_NODE 4 CAST
DATA_NODE 5 MUL integer
CONST_NODE 2 4
DATA_NODE 6 CAST
DATA_NODE 7 ADD integer
DATA_NODE 8 LOAD events
DATA_NODE 9 CAST
DATA_NODE 10 MUL integer
CONST_NODE 3 4

DATA_NODE 11 CAST
DATA_NODE 12 ADD integer
DATA_NODE 13 CAST
DATA_NODE 14 MUL integer
CONST_NODE 4 4
DATA_NODE 15 CAST
DATA_NODE 16 ADD integer
DATA_NODE 17 LOAD events
DATA_NODE 18 STORE events
DATA_NODE 19 CAST
DATA_NODE 20 MUL integer
CONST_NODE 5 4
DATA_NODE 21 CAST
DATA_NODE 22 ADD integer
DATA_NODE 23 STORE events
DATA_NODE 24 RETURN
DATA_CON 0 1 d a
DATA_CON 0 1 c b
DATA_CON 2 3 d a
DATA_CON 1 3 c b
DATA_CON 1 4 d a
DATA_CON 4 5 d a
DATA_CON 2 5 c b
DATA_CON 5 6 d a
DATA_CON 6 7 d a
DATA_CON 7 8 d a
DATA_CON 1 9 d a
DATA_CON 9 10 d a
DATA_CON 3 10 c b
DATA_CON 10 11 d a
DATA_CON 11 12 d a
DATA_CON 3 13 d a
DATA_CON 13 14 d a
DATA_CON 4 14 c b
DATA_CON 14 15 d a
DATA_CON 15 16 d a
DATA_CON 16 17 d a
DATA_CON 12 18 d a
DATA_CON 17 18 d d
DATA_CON 3 19 d a
DATA_CON 19 20 d a
DATA_CON 5 20 c b
DATA_CON 20 21 d a
DATA_CON 21 22 d a
DATA_CON 22 23 d a
DATA_CON 8 23 d d

This format enumerates all nodes and edges. Nodes can be data nodes or constant nodes. An edge connects two data nodes or a constant node and a data node. Multiple Basic Blocks are also supported. IRG produces a visual depiction of the code above, showing data connections, operator types, constants, etc. Fig. 38 shows the Alter stage represented in this graphical format.

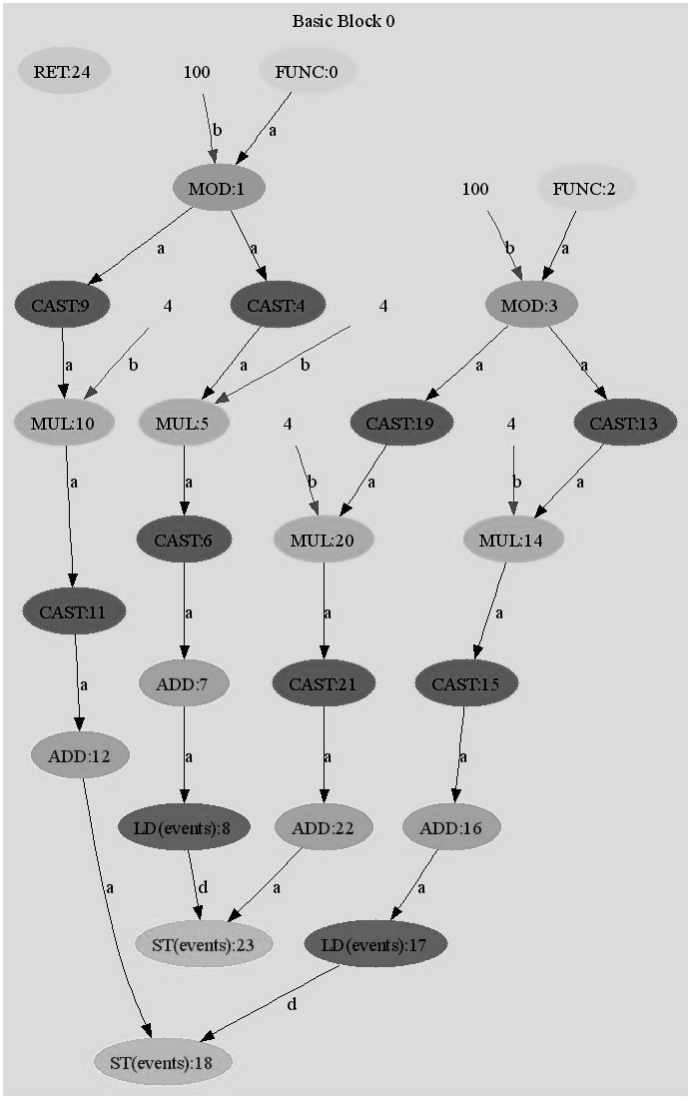


Figure 38: Raw intermediate representation of the Alter stage.

The format depicted in fig. 38 is a direct translation from the gcc file. There are significant optimizations and simplifications that can be made. For example, fig. 38 depicts only data dependencies, not anti-dependencies. Also, several nodes and sub-graphs can be removed. Node 24 is a return node (RET). This has no bearing on the graph functionality and is thus discarded. Array indexing or any sort of address computation can also be simplified. Because gcc maps 32-bit integer data onto a byte-addressable processor, a cast – multiply (by four) – cast – add (base + offset) construct appears when an array element is addressed. This entire sub-graph can be removed, as

the target architecture memory is word-addressable and the base of the array is always at address zero. Fig. 39 depicts the optimized Alter graph upon exit from the IRG stage.

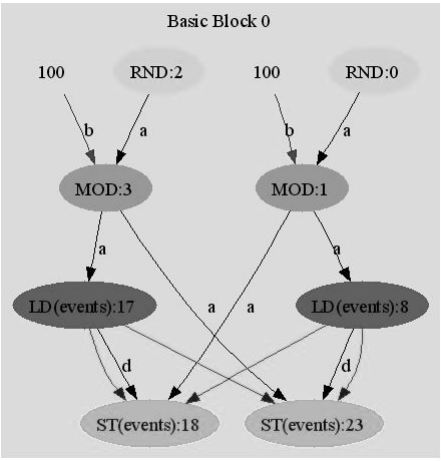


Figure 39: Optimized intermediate representation of the TSP Alter function.

The intermediate representation consists of arithmetic and data-transfer nodes. Fig. 39 contains random number generation (RND), modulo (MOD), load (LD), and store (ST) instructions. Constants are also represented as nodes. Edges represent data dependencies and anti-dependencies. Data dependency edges are labeled according to their relationship to the node they feed. For example, a MOD node needs an input labeled "a" and one labeled "b". A store node needs an address to write to (labeled "a") and data to write (labeled "b"). Unlabeled edges represent anti-dependencies. Similar graphs are produced for each of other three steps in the pipeline and the Adjust Temperature module.

5.3 Intermediate Format to Architecture

Once the different functions have been parsed and converted to intermediate format, SATH must perform a mapping of intermediate format to template for each stage. As shown in fig. 37, different stages in the pipeline achieve this mapping in different manners. The architectures of the Copy, Alter, and Adjust Temperature processors are fixed. Regardless of the specifics of the simulated annealing algorithm under conversion, these three modules perform the same function. The Copy Processor copies a solution from one memory bank to another, word by word. The solution size, which is the number of words to copy, is extracted as part of the Copy Constant Extraction block shown in fig. 37. The Adjust Temperature Constant Extraction block extracts the initial temperature and the cooling rate.

The internals of the Alter and Evaluate processors, on the other hand, vary with differing applications. The general form of each stage, however, can be characterized. An Alter Processor can generally be characterized as a modification to one or two entries in the

solution. In the case of the TSP example, two locations in the array are selected at random and the data is swapped, as shown in fig. 39. This changes the order in which cities are visited and will affect the score of the solution.

An architecture which supports the CDFG shown in fig. 39 is shown in fig. 40. It consists of one random number generator, one modulo unit, one read port, and one write port, with additional multiplexers and delay registers to provide proper timing. Additionally, a control unit is needed to manage multiplexer select and memory write enable lines. This architecture is optimized for resource utilization. Because functional units such as the random number generator and modulo units are used twice, the latency of this architecture is larger than that of an architecture with multiple RND and MOD units.

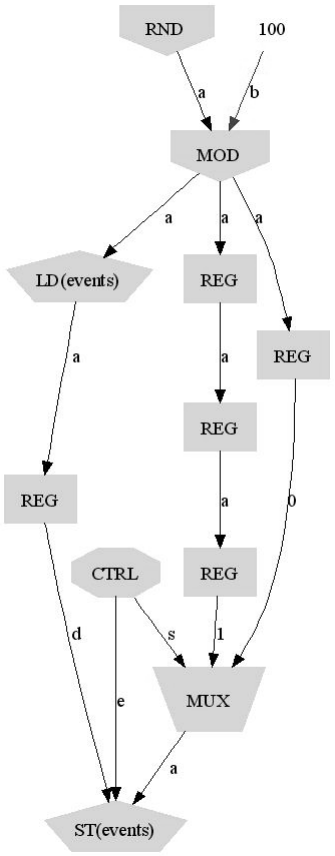


Figure 40: Minimal architecture for the TSP Alter function.

The Evaluate processor, on the other hand, is generally the most compute-intensive stage in the pipeline. C code for the Evaluate function of the example TSP problem is shown below.

```
distance = 0;
for (i=0; i<MAX_EVENTS-1; i++) {
    distance += (abs(x_pos[next[i]] – x_pos[next[i+1]]) +
            abs(y_pos[next[i]] - y_pos[next[i+1]]));
}
```

The intermediate representation of this Evaluate function is shown in fig. 41.

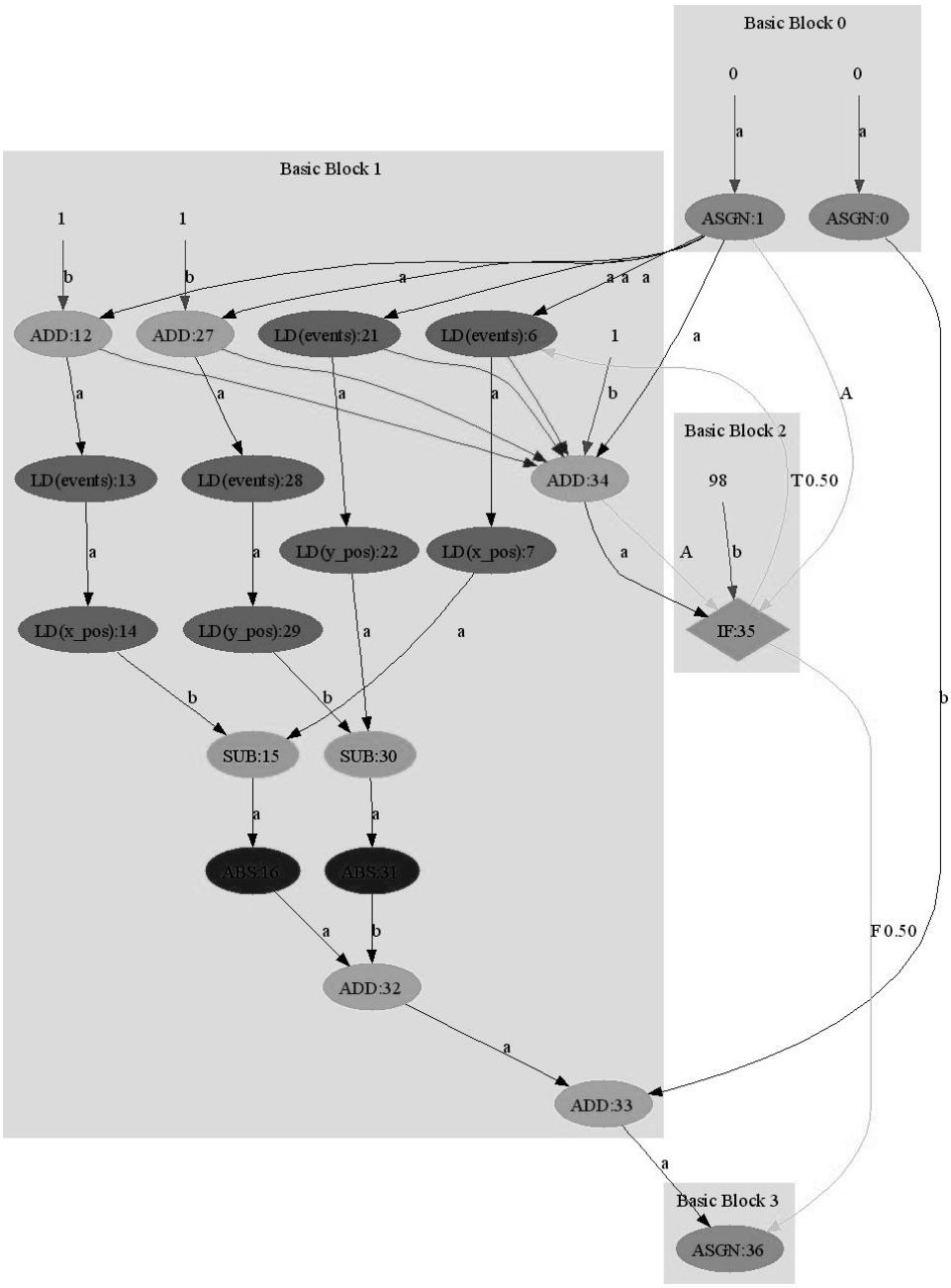


Figure 41: CDFG of the TSP Evaluate function.

Evaluate functions generally look at every element in a solution at least once and compute a score based upon relationships between elements. In the case of TSP, the Manhattan distance must be computed between the current city and the next city to be visited for every city in the schedule. Because the "for" loop representing such a code has no loop-carried dependencies, a pipelined architecture, consisting of the loop internals, can be derived for efficiently computing such an architecture. To derive an efficient pipeline that can accept new data on every clock cycle, there must be a one-to-one mapping between each node in the CDFG and a corresponding module in the

architecture. Fig. 42 shows the minimal pipeline architecture for the Evaluate Processor of the TSP problem.

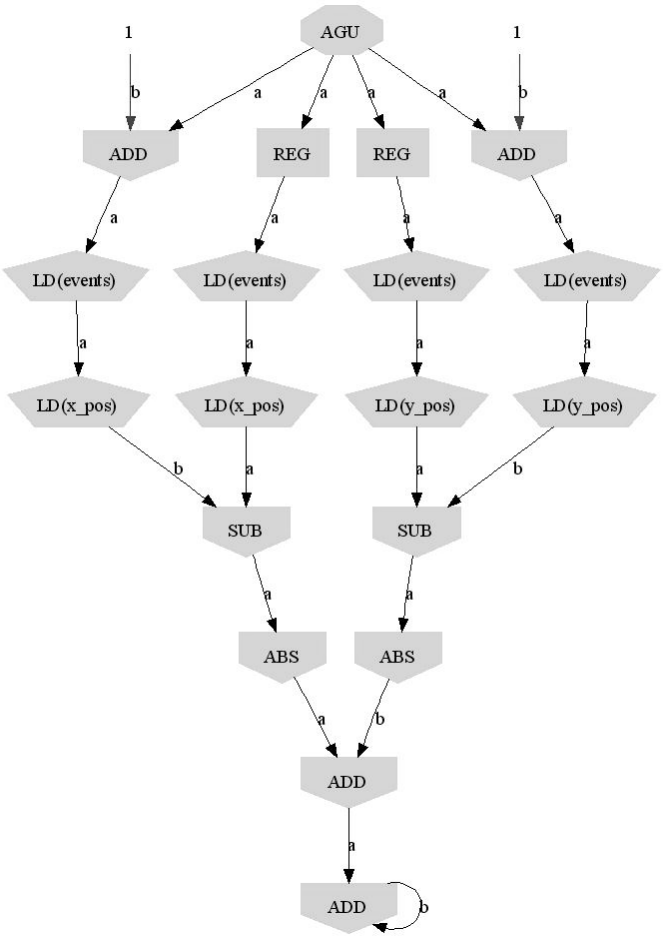


Figure 42: Pipelined architecture of the TSP Evaluate function.

As the general target architecture is a coarse-grained pipelined processor, solutions can only be passed between stages at a rate equal to the latency of the slowest stage. Because of this, careful analysis must be taken to ensure that the latency of each stage is reduced as much as possible. A global DSE algorithm, shown in fig. 43, is employed to derive this pipelined architecture.

```
generate minimal architecture for all stages
compute latency of all stages
Loop
     identify worst stage latency
     reduce latency of worst stage through DSE by either
          a.   reducing latency enough to pass "worst
               stage" label to a different stage OR
          b.   reducing latency as much as possible
               while retaining "worst stage" label
EndLoop (when worst stage label cannot be passed)
```

Figure 43: DSE algorithm for generating a pipelined processor.

As the Copy, Alter, and Adjust Temperature processor architectures (and associated latencies) are fixed, the algorithm for exploring possible Alter and Evaluate processors

proceeds as follows. First, a minimal architecture is derived for both the Alter and Evaluate processing stages. For the TSP example, these are the architectures shown in figures 40 and 42, respectively. The latencies of the architectures are determined by using Ripple List Scheduling, described in Chapter 4, to schedule CDFG nodes on available resources. The Xilinx Virtex 4 SX35 FPGA resource utilization of the architectures, in terms of slices, block RAMs, and DSP48 units is also estimated, using the estimation techniques described in Chapter 4.

Once the initial latency and resource utilization for both the Alter and Evaluate stages has been computed, the algorithm enters the optimization loop. The loop consists of two steps. First, the worst-performing stage of the pipeline is identified as the current candidate for improvement. If this stage is the Copy, Accept, or Adjust Temperature stage, nothing can be done and the algorithm terminates. However, if the worst-performing stage is Alter or Evaluate, DSE is performed in an attempt to reduce execution latency through exploitation of available parallelism. Because of the architectural differences between the Alter and Evaluate stages, two different types of DSE are employed. For the simpler Alter Processor, instruction-level parallelism is explored. For the minimal Alter architecture, only one instance of each needed type is allowed. Potential increases in performance can be explored by providing additional random number generation, modulo, load, and/or store resources to the architecture. As the number of combinations of different resources is large, a simulated annealing algorithm is used to perform this exploration, repeatedly measuring the tradeoff between latency and resource utilization. Note that the goal, according to fig. 43, is not to find the fastest implementation, but only an implementation that allows the label of "worst stage" to be passed to another stage. This prevents the introduction of unneeded, costly architectural features that wouldn't improve the pipeline performance.

A DSE technique for the Evaluate Processor is different. Because of the loop structure and pipelined architecture associated with this stage, loop unrolling is performed, rather than instruction-level analysis. The architecture shown in fig. 42 can be duplicated and half of the work can be sent to each instance. This more than doubles the resource utilization, as additional glue logic is needed to combine the results of each half at the end. No simulated annealing algorithm is needed in this case, as the number of different circuits is small. It would be infeasible to loop unroll an Evaluate architecture more than six or eight times, because of the associated resource utilization and overhead. Because of this, loop unrolling in SATH is capped at eight times.

Once the worst stage has been bettered, the loop repeats until no improvements can be made. The DSE algorithm terminates on one of three conditions. Either (1) the Copy, Accept, or Adjust Temperature processor is the worst performer; (2) the Evaluate or Alter stage is the worst performer, but no additional parallelism can be extracted to improve performance; or (3) the Evaluate or Alter stage is the worst performer, but there are not sufficient resources remaining to allow for additional extraction of parallelism.

One additional item worth mentioning is that the data width of the processor is specified by the user, conserving computation time and resources that would be wasted if data

widths were restricted to the traditional 16- or 32-bit words. The user must determine what data widths are appropriate for a given application.

5.4 Architecture to VHDL

Once an architecture has been finalized, it must be translated into hardware. Hardware Description Languages (HDLs) such as Verilog and/or VHDL are commonly used to construct both simple and complex hardware modules. The HDLs allow the creation of reusable models, but the re-usability of a design does not depend on language features alone. It requires design discipline to reach an efficient reusable design. This section describes a VHDL generation tool which produces synthesizable VHDL code for an entire circuit, based upon an input text file, which contains a description of the functional units and the various interconnections between them.

The VHDL generator consists of two primary blocks, (1) The Architecture parser/Translator which parses the architecture description file for architectural units like adders and multiplexers, and instantiates modules as necessary and (2) The Memory Generator which generates memory-based controllers for the select lines of the multiplexers present in the circuit.

The hardware specifications mentioned in the architecture description file should be sufficient to produce the required hardware in VHDL. The text format should be such that it supports all possible combinations of hardware units without any discrepancies. The architecture file is capable of giving details of the hardware units and all possible interconnections for the VHDL generation tool. Each line of the text file specifies either a hardware unit or an interconnection between hardware units. Consider as an example the circuit shown in fig. 44.

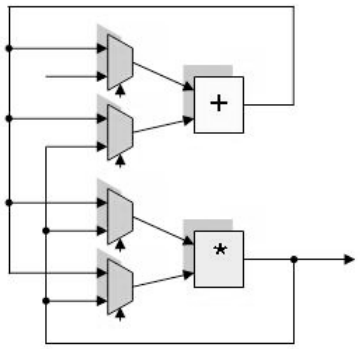


Figure 44: A simple circuit consisting of an adder, a multiplier, and four multiplexers.

The input text file for this architecture looks like this:

```
u0      INT_ADD     16
u1      INT_MUL     16
u2      MUX  16     2
u3      MUX  16     2
u4      MUX  16     2
u5      MUX  16     2
```

b0	16	u2	3	u0	0						
b1	16	u3	3	u0	1						
b2	16	u4	3	u1	0						
b3	16	u5	3	u1	1						
b4	16	u1	2	u3	1	u4	1	u5	1	out	0
b5	16	u0	2	u2	0	u3	0	u4	0	u5	0
b6	16	in	0	u2	1						
b7	2	sel	1	u2	2						
b8	2	sel	2	u3	2						
b9	2	sel	3	u4	2						
b10	2	sel	4	u5	2						

In this representation, a "u" denotes a hardware unit in the circuit and a "b" denotes an interconnection (bus). The hardware unit on the first line is defined by a unit number (u0), followed by the unit type (INT_ADD) and the data width (16). For a multiplexer, data width and number of inputs must be specified. A bus is defined by a bus number (b0), a data width (16), a source unit (u2) and port (3), and a destination unit (u0) and port (0). A bus must have only one source unit, but may include multiple destination units.

When a particular dependency graph is mapped to a particular set of resources, a state machine must be created to synchronize the flow of data through the circuit by controlling the multiplexer select line. The VHDL generator takes the values of the select lines for each time step from an input text file, generates a memory module, and connects the module to the select lines of the multiplexers in the hardware implementation.

The input text file for this memory takes the following format:

mux_8	mux_4	mux_4
7	1	3
3	1	0
1	0	2
5	1	1
6	7	8

In this representation, "mux_8" denotes a multiplexer with 8 inputs. The second row gives the values for the first clock cycle to the input select line of the multiplexer. The next row gives the value of the select line input in the next clock cycle, and so forth.

6. RESULTS

The architectural derivation and performance for the TSP example discussed throughout this chapter are now be detailed. The C code is input to SATH, along with the parameters listed in table 18 for the static modules. 16-bit data widths are used for this design. The first step in the algorithm outlined in fig. 33 is to derive a minimal architecture for each stage. The Copy, Accept, and Adjust Temperature stages have a fixed architecture with the resource utilization and performance listed in Table 18.

Table 18: Input parameters for TSP example.

Parameter	Value
V4SX35 Total Slices	15,360 +192 BRAMs
Main Controller Slice Usage	177
Memory Slice Usage	1,035 + 20 BRAMs
Copy Slice Usage	33
Accept Slice Usage	1210 + 1 BRAM
Adj. Temp. Slice Usage	205 + 4 DSP48s
Copy Latency	101
Accept Latency	54
Adj. Temp. Latency	12

The initial architectures for the Alter and Evaluate stages are those shown in fig. 40 and 42, respectively. The initial resource usage of the Alter stage is 335 slices with a latency of 24 cycles. The initial Evaluate stage uses 66 slices and 4 BRAMs and has a latency of 106 cycles.

Now that the initial architecture resources and times have been computed, the algorithm enters the improvement loop. The “worst stage” label currently belongs to the Evaluate Processor, as its 106-cycle latency is worse than any that of any other stage. As this is the Evaluate stage, improvement is introduced by unrolling the loop once, resulting in the architecture shown in fig. 45.

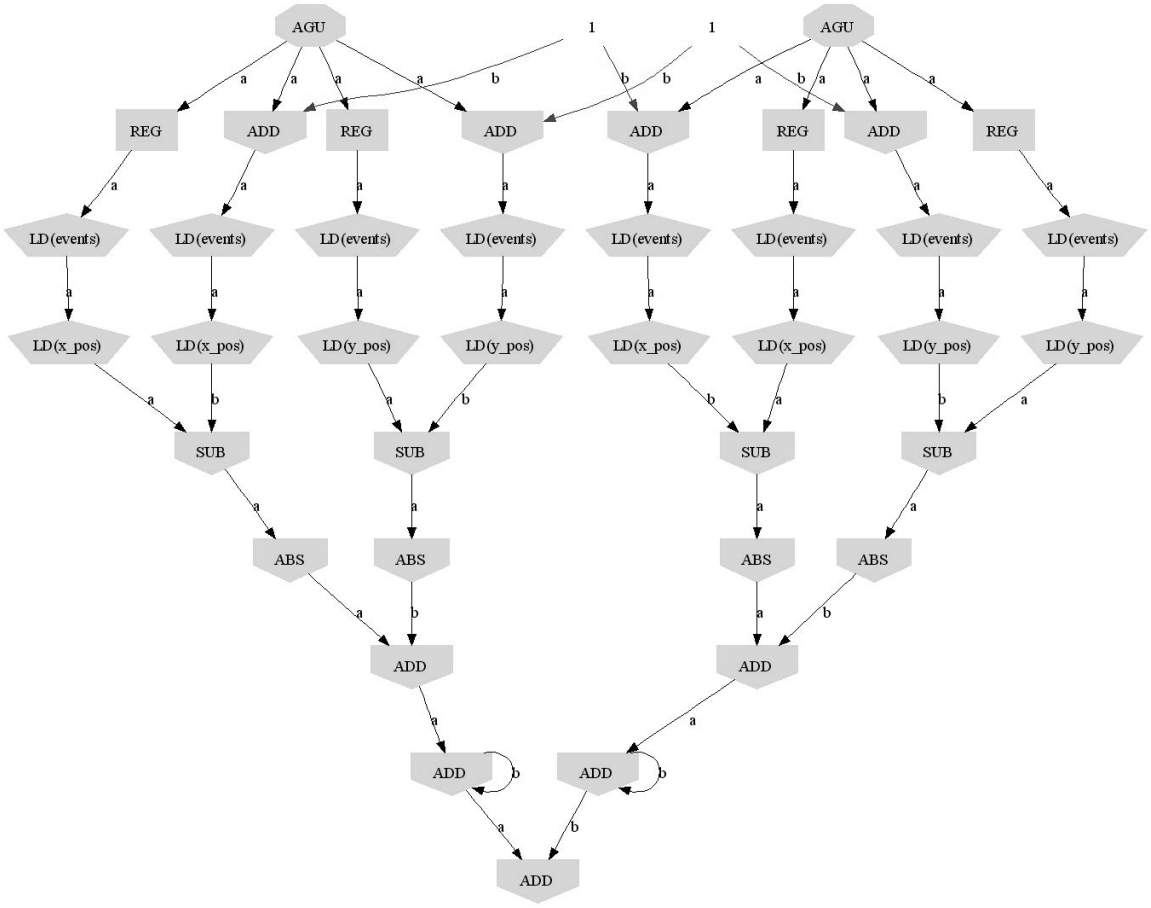


Figure 45: Dual pipeline architecture (one loop unroll) of the TSP Evaluate function.

Notice that the basic architecture shown in fig. 42 is repeated and an additional adder is included to combine the results from the two pipelines. The latency of this version of the architecture is 54 cycles, with a resource utilization of 140 slices and eight BRAMs. There is ample room on the FPGA to support this improved functionality.

Once this step is complete, the "worst stage" flag passes from the Evaluate Processor to the Copy Processor. Condition (1) for termination is now valid (the Copy, Accept, or Adjust Temperature processor is the worst performer) and thus the architecture is complete.

The final step is to determine how the performance of the application-specific processor derived by SATH compares with that of the same C code running on both a traditional desktop processor and a widely used embedded PowerPC (architecturally similar to those flown in space). The C code specifies an initial temperature of 10,000, a (geometric) cooling rate of 0.9999, and a stop threshold of 0.0001, meaning that the simulated annealing loop executes 184,198 times. Table 19 shows a performance comparison between the SATH-produced TSP ASIP, the TSP code running on an AMD Athlon 64 X2 Dual Core Processor running at 2.61 GHz with 1 GB of RAM, and a PowerPC 405 processor.

Table 19: Speedup comparison for SATH-generated architecture.

	Clock Cycles	Clock Freq.	Execution Time	Speedup
PowerPC Processor	1.72×10^9	100 MHz	17.2 s	1.0
Athlon Processor	1.26×10^9	2.61 GHz	483 ms	35.6
SATH ASIP	18.6×10^6	150 MHz	124 ms	138.7

From table 19, it can be seen that the custom architecture performs 138 times better than the same algorithm running on a PowerPC. The ASIP also outperforms a desktop PC by a factor of 3.9. The reasons for this speed-up are three-fold. First, the custom circuit employs a four-stage macro pipeline. This allows for four different solutions to be at different stages of processing simultaneously, rather than only managing one solution at a time. Second, the most complex of the processing stages, the evaluate function, has been loop-unrolled in the custom implementation to decrease the latency of the pipeline. Third, in a conventional processor, up to 50 percent of the computation cycles in typical applications can be consumed by load and store instructions, especially in CISC architectures which have few internal registers. Because of the application-specific nature of the SATH approach, no unneeded load/store cycles are consumed.

The performance gain for this example is very conservative when compared to gains for a more complicated piece of code, such as that described in Chapter 3. In general, additional speedup can be obtained when the Evaluate function consists of a score that is the sum of multiple independent parts that can be computed in parallel.

REFERENCES

[1] A. Winterholler, M. Roman, D. Miller, J. Krause, and T. Hunt, "Automated core sample handling for future Mars drill missions," in *8th International Symposium on Artificial Intelligence, Robotics and Automation in Space* Germany, 2005.

[2] "New Space Communications Capabilities Available for NASA's Discovery and New Frontier Programs," in *NASA Technology Discovery/New Frontier Roadmap*, 2006.

[3] S. Knight, G. Rabideau, S. Chien, B. Engelhardt, and R. Sherwood, "Casper: space exploration through continuous planning," *IEEE Intelligent Systems,* vol. 16, pp. 70-75, 2001.

[4] S. A. Edwards, "The challenges of hardware synthesis from C-like languages," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2005, pp. 66-67.

[5] C. E. Stroud, R. R. Munoz, and D. A. Pierce, "Behavioral model synthesis with Cones," *IEEE Transactions on Design and Test of Computers,* vol. 5, pp. 22-30, 1988.

[6] D. Ku and G. De Micheli, "HardwareC - A Language for Hardware Design," *CSL-TR-90-419,* 1990.

[7] D. Galloway, "The Transmogrifier C hardware description language and compiler for FPGAs," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1995, pp. 136-144.

[8] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*: Kluwer, 2002.

[9] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high speed ASICs," in *Proceedings of the Design Automation Conference*, 1998, pp. 315-320.

[10] R. J. Lipton, D. N. Serpanos, and W. H. Wolf, "PDL++: an optimizing generator language for register transfer design," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1990, pp. 1135-1138 vol.2.

[11] D. Soderman and Y. Panchul, "Implementing C algorithms in reconfigurable hardware using C2Verilog," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 339-342.

[12] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, Cyber," in *Proceedings on the Design, Automation and Test in Europe Conference and Exhibition*, 1999, pp. 390-393.

[13] Celoxica, "www.celoxica.com," *Handel-C Language Reference Manual,* 2003.

[14] C. Hoare, "Communicating sequential processes," *CACM,* vol. 21, pp. 666-667, 1978.

[15] T. Kambe, A. Yamada, K. Nishida, K. Okada, M. Ohnishi, A. Kay, P. Boca, V. Zammit, and T. Nomura, "A C-based synthesis system, Bach, and its application," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2001, pp. 151-155.

[16] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*: Kluwer, 2000.
[17] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2005, pp. 317-322.
[18] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *Proceedings of the 16th International Conference onVLSI Design*, 2003, pp. 461-466.
[19] M. Budiu and S. Goldstein, "Compiling application-specific hardware," in *Proc. FPL, LNCS*, Montpellier, France, 2002, pp. 853-863.
[20] S. A. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages," *IEEE Design and Test of Computers,* vol. 23, pp. 375-386, 2006.
[21] K. Hong-Soog, Y. Young-Ha, N. Sang-Og, and H. Dong-Soo, "ICU-PFC: an automatic parallelizing compiler," in *Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 2000, pp. 243-246 vol.1.
[22] H. Kasahara, M. Obata, K. Ishizaka, K. Kimura, H. Kaminaga, H. Nakano, K. Nagasawa, A. Murai, H. Itagaki, and J. Shirako, "Multigrain automatic parallelization in Japanese Millennium Project IT21. Advanced Parallelizing Compiler," in *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, 2002, pp. 105-111.
[23] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, L. Shih-Wei, and E. Bu, "Maximizing multiprocessor performance with the SUIF compiler," *Computer,* vol. 29, pp. 84-89, 1996.
[24] M. B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim, "Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004, pp. 2-13.
[25] G. Mehta, R. R. Hoare, J. Stander, and A. K. Jones, "Design space exploration for low-power reconfigurable fabrics," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006, p. 4 pp.
[26] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Comparing Three Heuristic Search Methods for Functional Partioning in Hardware-Software Codesign," *Design Automation for Embedded Systems,* vol. 6, pp. 425-449, 2002.
[27] B. Miramond and J. M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2005, pp. 366-371 Vol. 1.
[28] C. Talarico, E. Rodriguez-Marek, and K. Min-sung, "Multi-objective design space exploration methodologies for platform based SOCs," in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, 2006, p. 7 pp.

[29] C. Silvano, G. Agosta, and G. Palermo, "Efficient architecture/compiler co-exploration using analytical models," *Design Automation for Embedded Systems,* vol. 11, pp. 1-23, 2007.
[30] K. Kurt, R. Kaushik, S. Nadathur, and J. Yujia, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, 2005, pp. 273-278.
[31] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. Ozturan, and G. Diindar, "Optimizing Instruction-set Extensible Processors under Data Bandwidth Constraints," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2007, pp. 1-6.
[32] B. So, M. Hall, and P. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *Conference on Programming Language Design and Implementation*, Germany, 2002, pp. 165-176.
[33] S. Bilavarn, G. Gogniat, J. L. Philippe, and L. Bossuet, "Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 25, pp. 1950-1968, 2006.
[34] C. Silvano, D. Sciuto, D. Bruschi, and G. Beltrame, "Decision-theoretic exploration of multiProcessor platforms," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, 2006, pp. 205-210.
[35] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *Computers and Digital Techniques, IEE Proceedings-,* vol. 153, pp. 173-180, 2006.
[36] G. Ascia, V. Catania, and M. Palesi, "Design space exploration methodologies for IP-based system-on-a-chip," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2002, pp. II-364-II-367 vol.2.
[37] V. Catania, G. Ascia, M. Palesi, D. Patti, and A. G. Di Nuovo, "Fuzzy decision making in embedded system design," in *Proceedings of the 4th international conference on hardware/software codesign and system synthesis*, 2006, pp. 223-228.
[38] E. M. d. Icaya, V. Rodellar, C. Gonzalez, V. Peinado, and V. Garcia, "Design Space Exploration for an Adaptive Noise Cancellation Algorithm," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs*, 2006, pp. 1-7.
[39] R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, and A. Fukunaga, "Using ASPEN to automate EO-1 activity planning," in *Proceedings of the IEEE Aerospace Conference*, 1998, pp. 145-152 vol.3.
[40] J. Holland, *Adaptation in Natural and Artificial Systems*: University of Michigan Press, 1975.
[41] S. Russel and P. Norvig, *Artificial Intelligence A Modern Approach*, 2nd ed.: Pearson, 2003.
[42] H. Emam, M. A. Ashour, H. Fekry, and A. M. Wahdan, "Introducing an FPGA based genetic algorithms in the applications of blind signals separation," in

Proceedings of the 3rd IEEE International Workshop onSystem-on-Chip for Real-Time Applications, 2003, pp. 123-127.

[43] M. S. Hamid and S. Marshall, "FPGA realisation of the genetic algorithm for the design of grey-scale soft morphological filters," in *Proceedings of the International Conference on Visual Information Engineering*, 2003, pp. 141-144.

[44] H. E. Mostafa, A. I. Khadragi, and Y. Y. Hanafi, "Hardware implementation of genetic algorithm on FPGA," in *Proceedings of the Twenty-First National Radio Science Conference*, 2004, pp. C9-1-9.

[45] S. D. Scott, A. Samal, and S. Seth, "HGA: A Hardware-Based Genetic Algorithm," in *Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays*, 1995, pp. 53-59.

[46] T. Wallace and Y. Leslie, "Hardware implementation of genetic algorithms using FPGA," in *Proceedings of the 47th Midwest Symposium on Circuits and Systems*, 2004, pp. I-549-52 vol.1.

[47] Z. Zhenhuan, D. Mulvaney, and V. Chouliaras, "Investigation Of A New Genetic Algorithm Designed For System-On-Chip Realization," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006, pp. 2981-2987.

[48] S. Narayanan and C. Purdy, "Hardware implementation of genetic algorithm modules for intelligent systems," in *Proceedings of the 48th Midwest Symposium on Circuits and Systems*, 2005, pp. 1733-1736 Vol. 2.

[49] T. Tachibana, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito, "General Architecture for Hardware Implementation of Genetic Algorithm," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 291-292.

[50] M. A. Vega-Rodriguez, R. Gutierrez-Gil, J. M. Avila-Roman, J. M. Sanchez-Perez, and J. A. Gomez-Pulido, "Genetic algorithms using parallelism and FPGAs: the TSP as case study," in *International Conference Workshops on Parallel Processing*, 2005, pp. 573-579.

[51] T. Anantharaman and R. Bisiani, "A hardware accelerator for speech recognition algorithms," in *Proceedings of the 13th annual international symposium on Computer architecture*, Tokyo, Japan, 1986, pp. 216-223.

[52] M. Wrighton and A. DeHon, "Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement," in *International Symposium on Field-Programmable Gate Arrays*, 2003, pp. 33-42.

[53] C. Shekhar, S. Raj, A. S. Mandal, S. C. Bose, R. Saini, and P. Tanwar, "Application Specific Instruction Set Processors: redefining hardware-software boundary," in *Proceedings of the 17th International Conference on VLSI Design*, 2004, pp. 915-918.

[54] K. Keutzer, S. Malik, and A. R. Newton, "From ASIC to ASIP: the next design discontinuity," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, pp. 84-90.

[55] F. Barat and R. Lauwereins, "Reconfigurable instruction set processors: a survey," in *Proceedings of the 11th International Workshop on Rapid System Prototyping*, 2000, pp. 168-173.

[56] B. R. Childers and J. W. Davidson, "An infrastructure for designing custom embedded counterflow pipelines," in *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, 2000, p. 10.
[57] M. K. Jain, M. Balakrishnan, and A. Kumar, "ASIP design methodologies: survey and issues," in *Proceedings of the Fourteenth International Conference on VLSI Design*, 2001, pp. 76-81.
[58] T. V. K. Gupta, R. E. Ko, and R. Barua, "Compiler-directed customization of ASIP cores," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002, pp. 97-102.
[59] V. P. Bhatt, M. Balakrishnan, and A. Kumar, "Exploring the number of register windows in ASIP synthesis," in *Proceedings of the 7th Asia and South Pacific and the 15th International Conference on VLSI Design*, 2002, pp. 233-238.
[60] G. Braun, A. Wieferink, O. Schliebusch, R. Leupers, H. Meyr, and A. Nohl, "Processor/memory co-exploration on multiple abstraction levels," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2003, pp. 966-971.
[61] G. Ascia, V. Catania, M. Palesi, and D. Patti, "Multiobjective optimization of a parameterized VLIW architecture," in *Proceedings of the NASA/DoD Conference on Evolvable Hardware*, 2004, pp. 191-198.
[62] S. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "Application-specific heterogeneous multiprocessor synthesis using extensible processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 25, pp. 1589-1602, 2006.
[63] O. Schliebusch, A. Chattopadhyay, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, and T. Kogel, "A framework for automated and optimized ASIP implementation supporting multiple hardware description languages," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2005, pp. 280-285 Vol. 1.
[64] T. Tascione, *Introduction to the Space Environment*, 2nd ed.: Krieger, 1994.
[65] M. Berg, "Fault Tolerance Implementation within SRAM Based FPGA Design Based upon the Increased Level of Single Event Upset Susceptibility," in *Proceedings of the 12th IEEE International On-Line Testing Symposium*, 2006, pp. 89-91.
[66] L. Yanmei, L. Dongmei, and W. Zhihua, "A new approach to detect-mitigate-correct radiation-induced faults for SRAM-based FPGAs in aerospace application," in *Proceedings of the IEEE National Aerospace and Electronics Conference*, 2000, pp. 588-594.
[67] C. A. Hulme, H. H. Loomis, A. A. Ross, and Y. Rong, "Configurable fault-tolerant processor (CFTP) for spacecraft onboard processing," in *Proceedings of the !EEE Aerospace Conference*, 2004, pp. 2269-2276 Vol.4.
[68] G. L. Smith and L. de la Torre, "Techniques to enable FPGA based reconfigurable fault tolerant space computing," in *Proceedings of the IEEE Aerospace Conference*, 2006, p. 11 pp.
[69] G. V. Larchev and J. D. Lohn, "Evolutionary based techniques for fault tolerant field programmable gate arrays," in *Proceedings of the Second IEEE International Conference on Space Mission Challenges for Information Technology*, 2006, p. 8 pp.

[70] T. L. Adam, K. Chandy, and J. Dickson, "A comparison of list scheduling for parallel processing systems," *Communications of the ACM,* vol. 17, pp. 685-690, 1974.

[71] M. Y. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, pp. 330-343, 1990.

[72] M. Shang, S. Sun, and Q. Wang, "An efficient parallel scheduling algorithm of dependent task graphs," in *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2003, pp. 595-598.

[73] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with inter processor communication times," *SIAM Journal on Computing,* vol. 5, pp. 879-886, 1994.

[74] K. Yu-Kwong and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems,* vol. 7, pp. 506-521, 1996.

[75] S. Govindarajan and R. Vemari, "Improving the schedule quality of static-list time-constrained scheduling," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2000, p. 749.

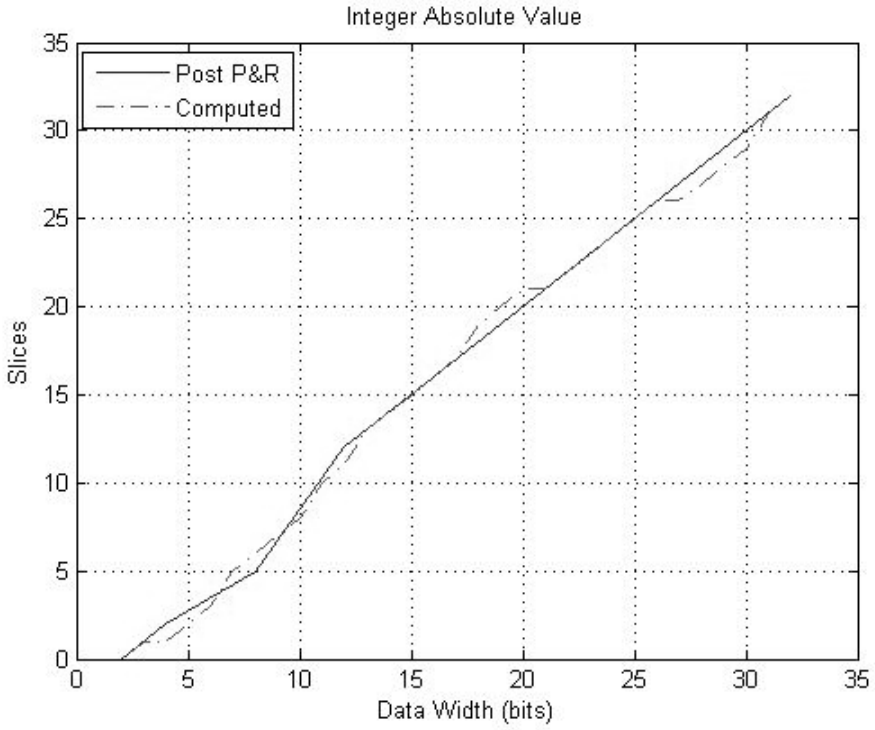
[76] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments," in *Proceedings of the International Conference on Parallel Processing Workshops*, 2003, pp. 149-155.

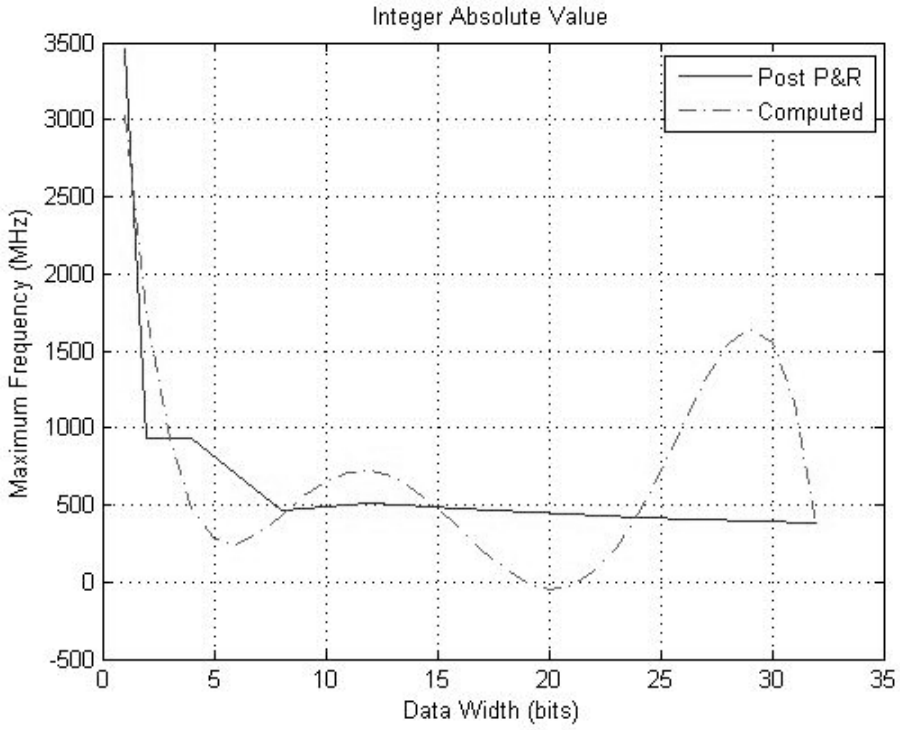
[77] S. Govindarajan and R. Vemuri, "Cone-based clustering heuristic for list-scheduling algorithms," in *Proceedings of European Design and Test Conference*, 1997, pp. 456-462.

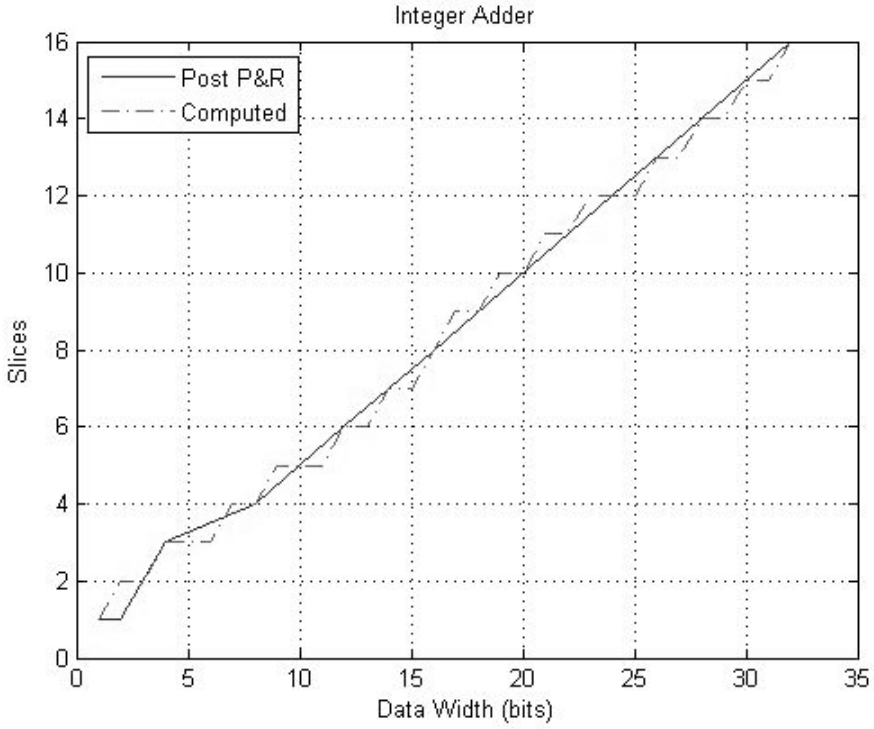
[78] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, "Scheduling of conditional process graphs for the synthesis of embedded systems," in *Proceedings of the Design, Automation and Test in Europe Conference*, 1998, pp. 132-138.

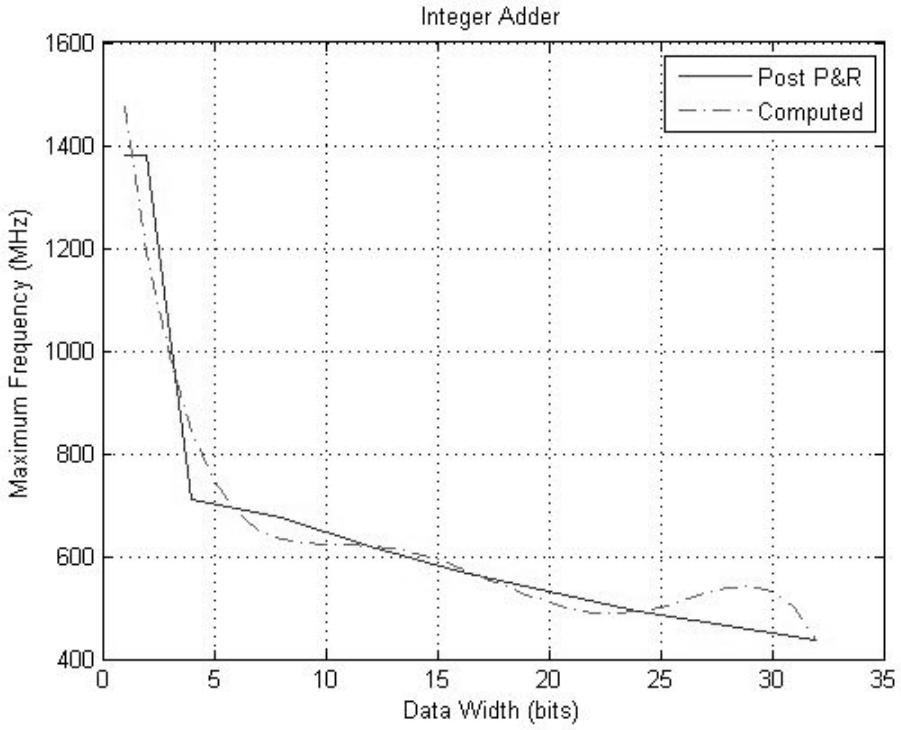
APPENDIX A

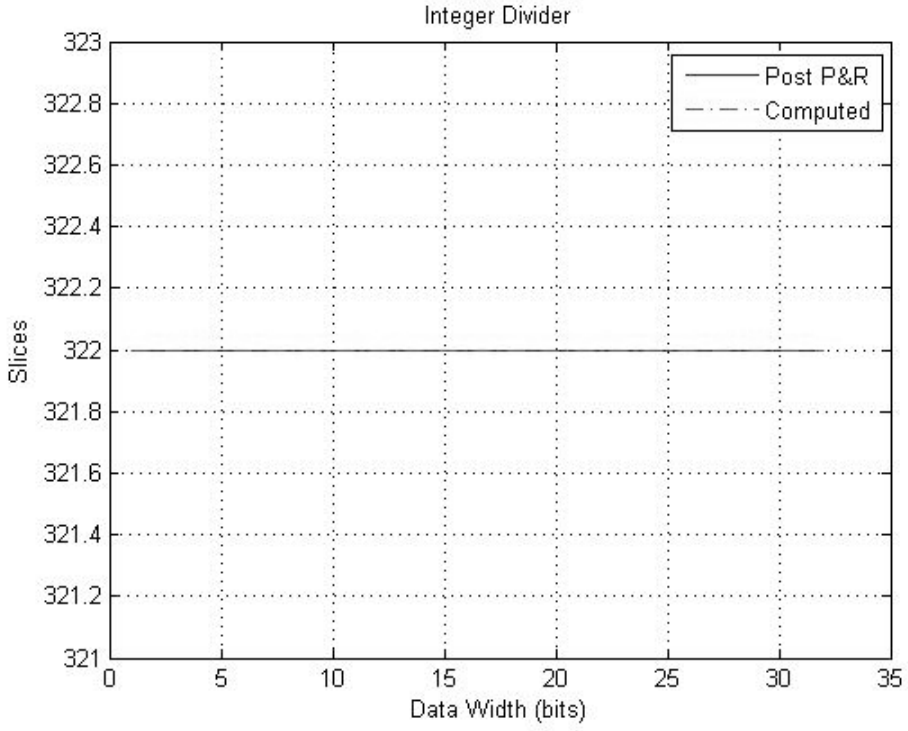
Resource Usage and Timing Constraints for Basic Building Blocks

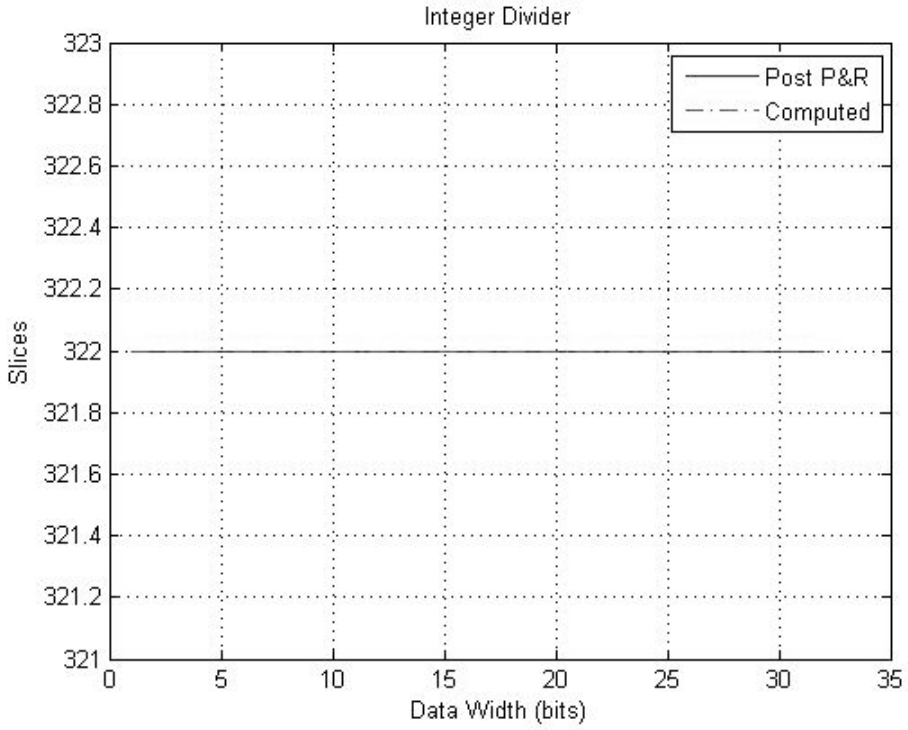


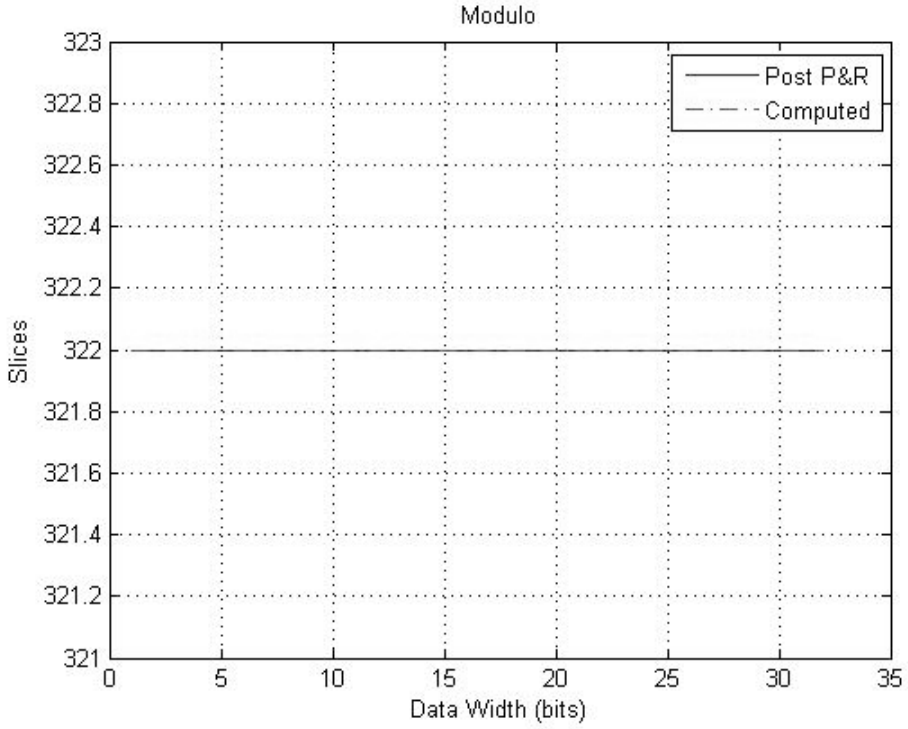

Integer Absolute Value
Post P&R
Computed
Maximum Frequency (MHz)
Data Width (bits)
3500
3000
2500
2000
1500
1000
500
0
-500
0
5
10
15
20
25
30
35

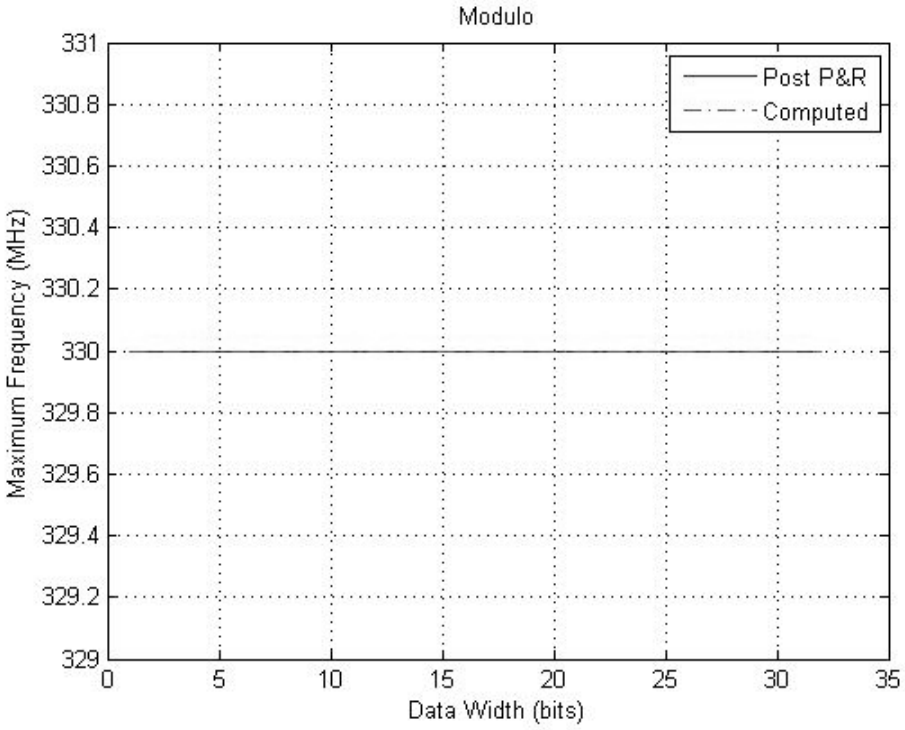

Integer Adder
Post P&R
Computed
Slices
Data Width (bits)
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35

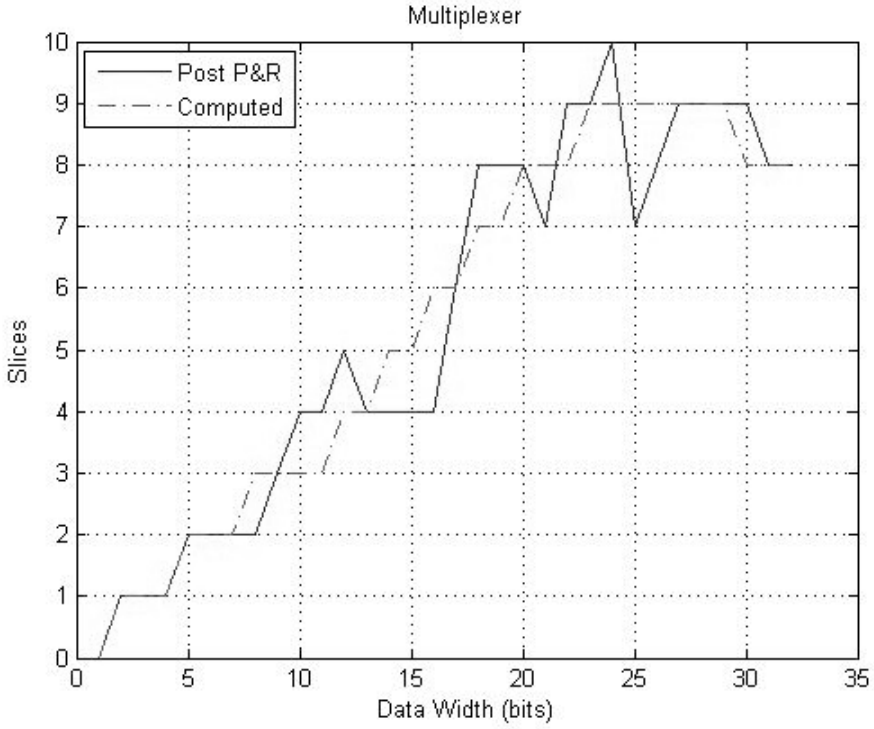

Integer Adder
Post P&R
Computed
Maximum Frequency (MHz)
Data Width (bits)
1600
1400
1200
1000
800
600
400
0
5
10
15
20
25
30
35

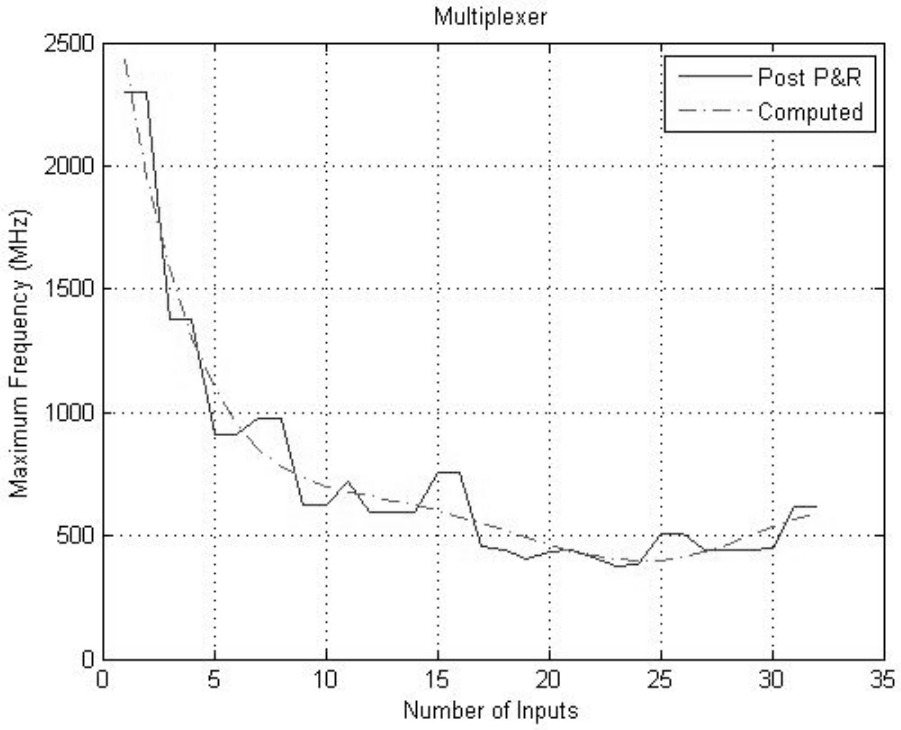

Integer Divider
Post P&R
Computed
Slices
Data Width (bits)
323
322.8
322.6
322.4
322.2
322
321.8
321.6
321.4
321.2
321
0
5
10
15
20
25
30
35

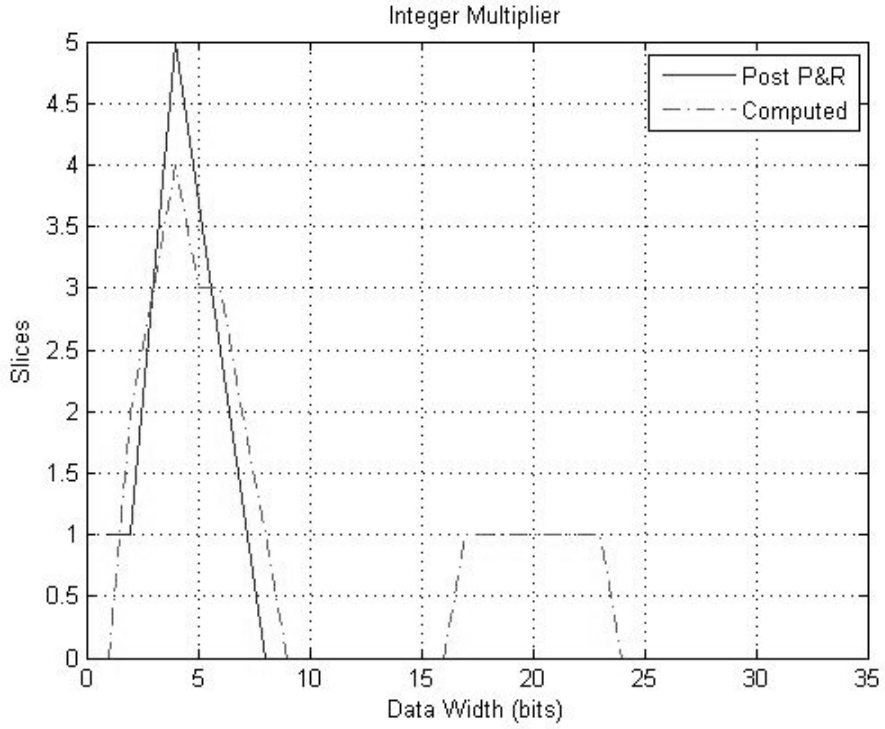

Integer Divider
Post P&R
Computed
Slices
Data Width (bits)
323
322.8
322.6
322.4
322.2
322
321.8
321.6
321.4
321.2
321
0
5
10
15
20
25
30
35

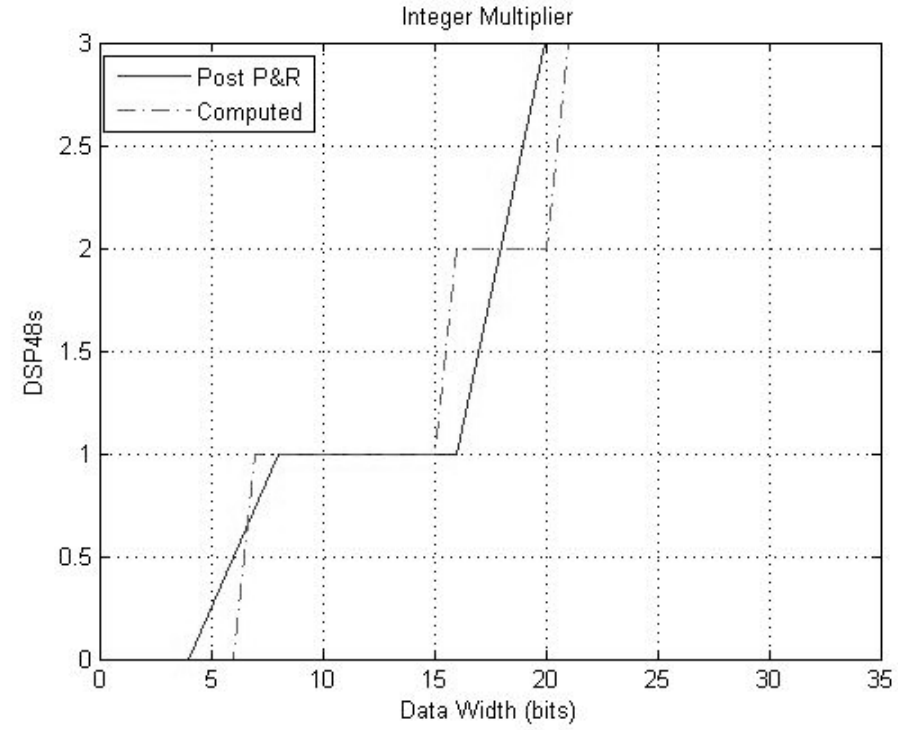

Modulo
Post P&R
Computed
Slices
Data Width (bits)
323
322.8
322.6
322.4
322.2
322
321.8
321.6
321.4
321.2
321
0
5
10
15
20
25
30
35

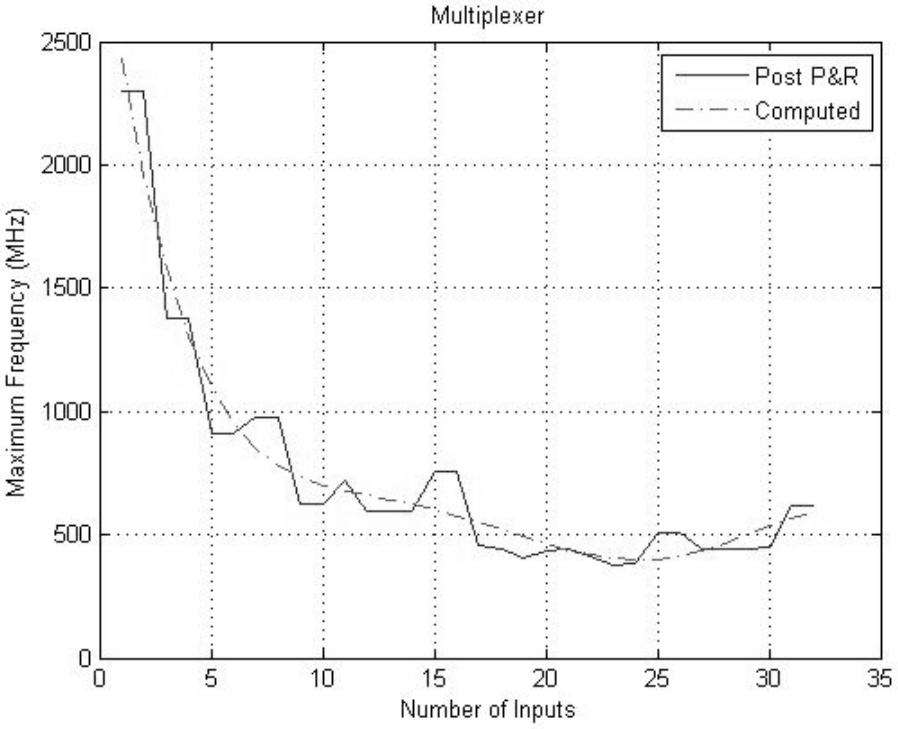

Modulo
Post P&R
Computed
Maximum Frequency (MHz)
331
330.8
330.6
330.4
330.2
330
329.8
329.6
329.4
329.2
329
0
5
10
15
20
25
30
35
Data Width (bits)

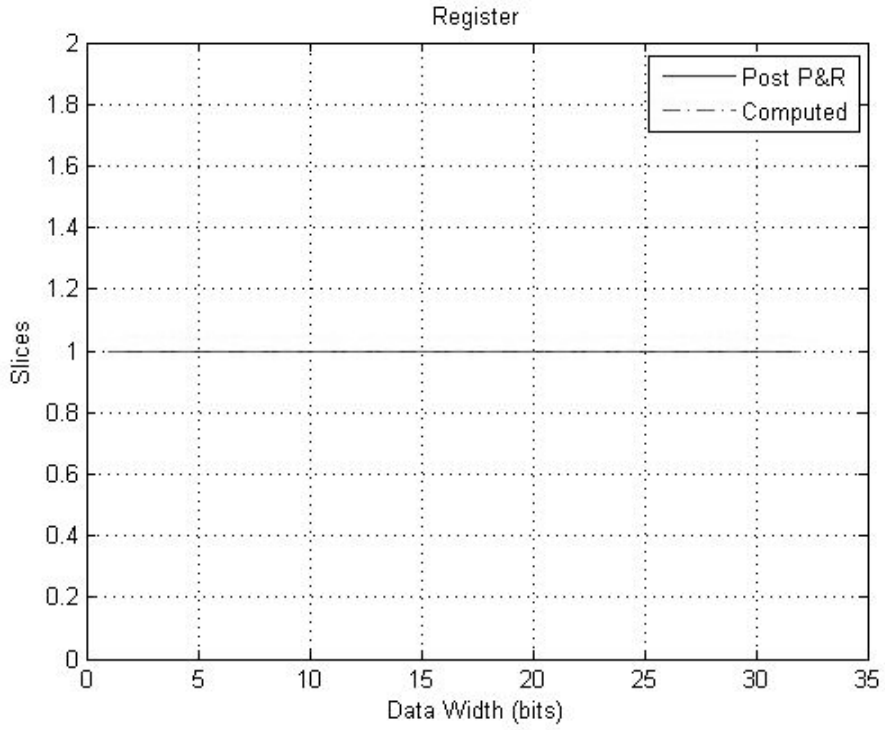

Multiplexer
Post P&R
Computed
Slices
10
9
8
7
6
5
4
3
2
1
0
0
5
10
15
20
25
30
35
Data Width (bits)

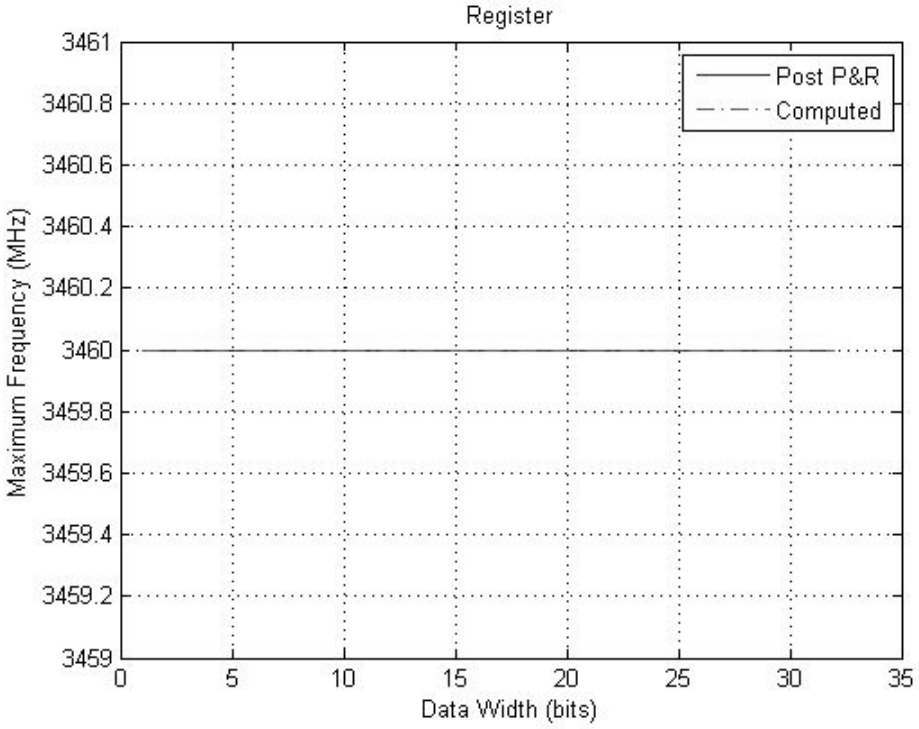

Multiplexer
Post P&R
Computed
Maximum Frequency (MHz)
Number of Inputs
0
500
1000
1500
2000
2500
0
5
10
15
20
25
30
35

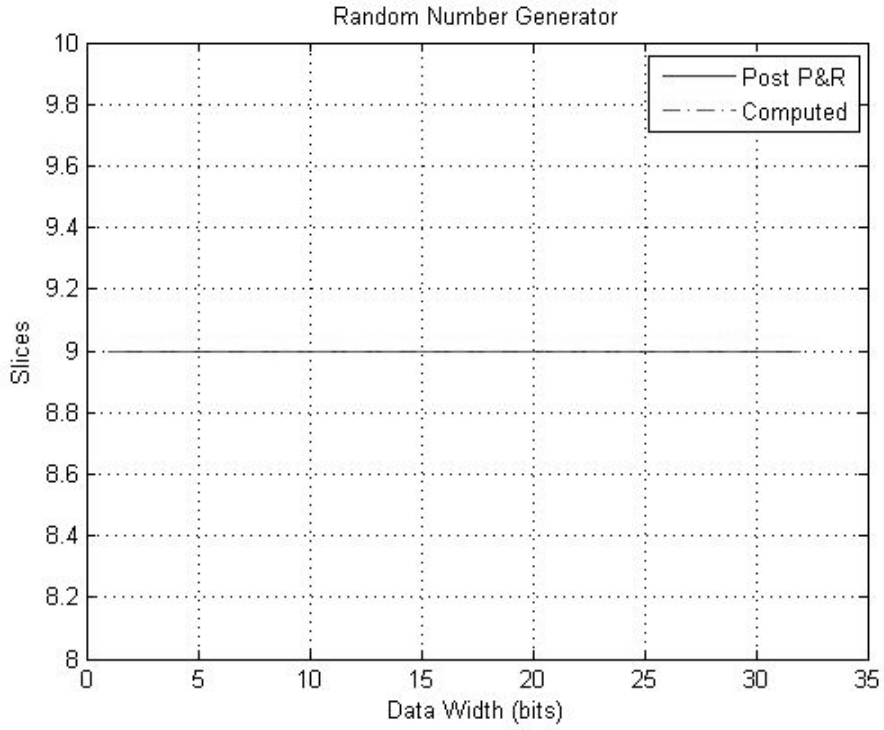

Integer Multiplier
Post P&R
Computed
Slices
Data Width (bits)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
5
10
15
20
25
30
35

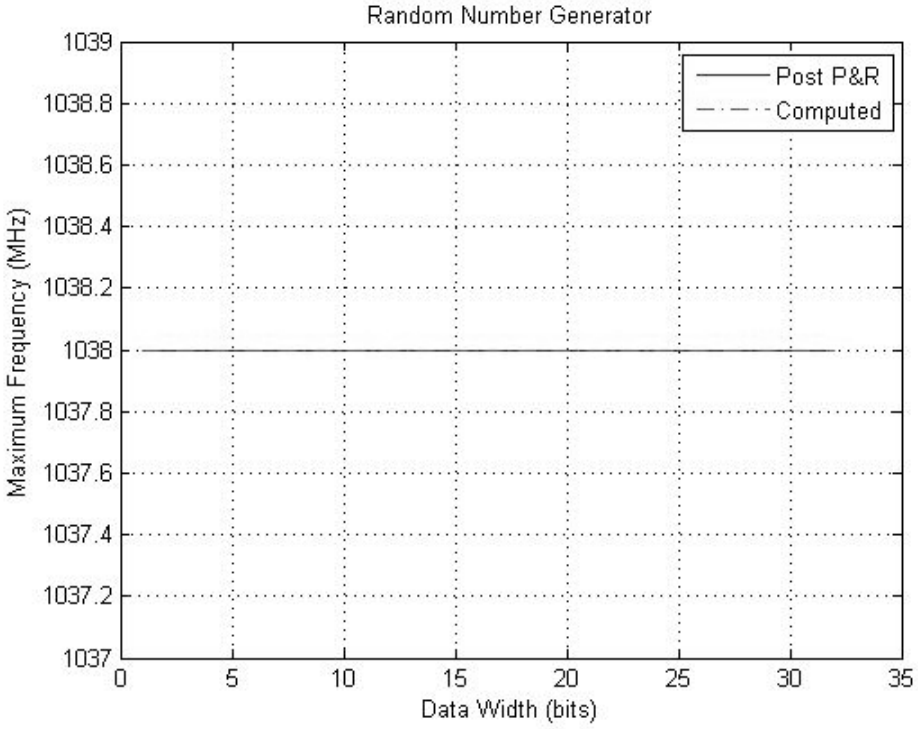

Integer Multiplier
Post P&R
Computed
DSP48s
Data Width (bits)
0
0.5
1
1.5
2
2.5
3
0
5
10
15
20
25
30
35

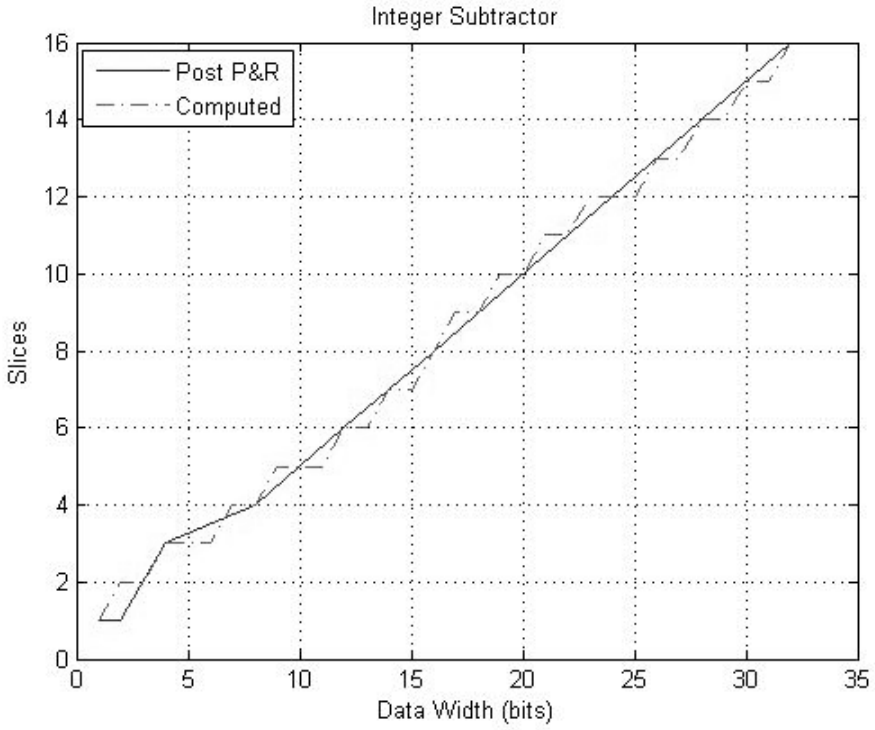

Multiplexer
Post P&R
Computed
Maximum Frequency (MHz)
Number of Inputs
0
500
1000
1500
2000
2500
0
5
10
15
20
25
30
35

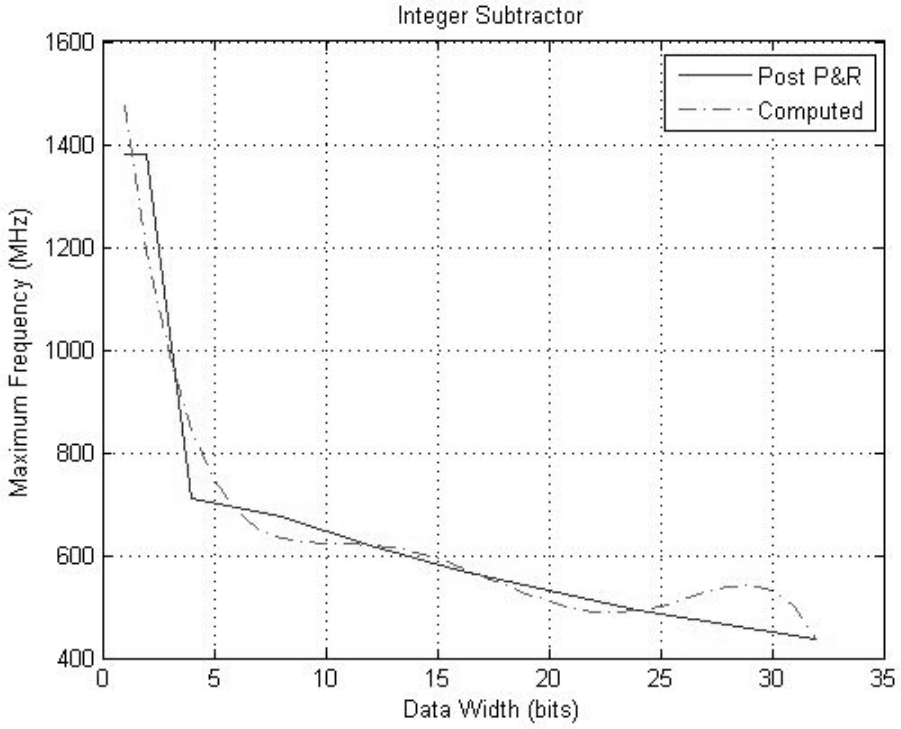

Register
Post P&R
Computed
Slices
Data Width (bits)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
5
10
15
20
25
30
35


Register
Post P&R
Computed
Maximum Frequency (MHz)
Data Width (bits)
3459
3459.2
3459.4
3459.6
3459.8
3460
3460.2
3460.4
3460.6
3460.8
3461
0
5
10
15
20
25
30
35


Random Number Generator
Post P&R
Computed
Slices
Data Width (bits)
10
9.8
9.6
9.4
9.2
9
8.8
8.6
8.4
8.2
8
0
5
10
15
20
25
30
35


Random Number Generator
Post P&R
Computed
Maximum Frequency (MHz)
Data Width (bits)
1039
1038.8
1038.6
1038.4
1038.2
1038
1037.8
1037.6
1037.4
1037.2
1037
0
5
10
15
20
25
30
35


Integer Subtractor
Post P&R
Computed
Slices
Data Width (bits)
0
2
4
6
8
10
12
14
16
0
5
10
15
20
25
30
35


Integer Subtractor
Post P&R
Computed
Maximum Frequency (MHz)
Data Width (bits)
400
600
800
1000
1200
1400
1600
0
5
10
15
20
25
30
35